Transformation semantics for a Model to Model (M2M) implementation

Auteur: Thaddé MAHMOURIAN
Directeur: Prof. Esteban ZIMANYI
Superviseur: Sabri SKHIRI, Euranova

Mémoire présenté en vue de l’obtention du titre d’ingénieur civil en informatique à finalité spécialisée en ingénierie informatique.

Année académique 2009-2010
# Table of Contents

1 Introduction 8  
1.1 Background .................................................................................. 8  
1.2 Motivation ..................................................................................... 9  
1.3 Structure ...................................................................................... 9  

2 Transformation: a state of the art 10  
2.1 XML Transformation ................................................................. 11  
2.1.1 XML ....................................................................................... 11  
2.1.2 XSL ....................................................................................... 12  
2.1.3 XPath ..................................................................................... 12  
2.1.4 XSLT ..................................................................................... 13  
2.1.5 XMI ....................................................................................... 16  
2.1.6 Model transformation with XMI and XSLT ......................... 17  
2.1.7 Altova XMLSpy<sup>TM</sup> ....................................................... 17  
2.2 XSLT elsewhere ......................................................................... 19  
2.3 ETL ......................................................................................... 20  
2.3.1 Talend Open Studio ............................................................. 21  
2.4 OLAP cubes ............................................................................. 22  
2.5 BPMN ....................................................................................... 23  
2.6 ATL ......................................................................................... 24  
2.6.1 Transformation directionality ............................................. 27  
2.6.2 Model synchronisation ........................................................ 27  
2.6.3 Atlas Model Weaver ............................................................. 27  
2.7 Conclusion ............................................................................. 29  

3 The Wazaabi Weaving model 30  
3.1 Introduction to EMF .................................................................. 30  
3.1.1 Ecore Model ........................................................................ 31  
3.1.2 Generator Model .................................................................. 31  
3.1.3 Code generation .................................................................. 31  
3.1.4 Editor .................................................................................. 33  
3.2 Wazaabi ............................................................................... 33  
3.3 Binding model ......................................................................... 34  
3.3.1 Location ............................................................................ 35  
3.3.2 Pointers ............................................................................. 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.3</td>
<td>Process</td>
<td>36</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Context</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>Using the Binding Model</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>Transformation triggering</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>Binding in Wazaabi UI</td>
<td>42</td>
</tr>
<tr>
<td>3.7</td>
<td>Conclusion</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>Analysis of the binding model</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Incremental transformation</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Building on Wazaabi</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>Transformation semantics</td>
<td>47</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Extending BPMN</td>
<td>50</td>
</tr>
<tr>
<td>4.4</td>
<td>Conclusion</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>Specification</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Purpose of the product</td>
<td>53</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Background</td>
<td>53</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Goals of the project</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Mandated constraints</td>
<td>54</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Solution constraints</td>
<td>54</td>
</tr>
<tr>
<td>5.2.2</td>
<td>“Off the shelf” packages</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>Naming conventions</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>Scope of the work</td>
<td>54</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Current situation</td>
<td>54</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Context of the work</td>
<td>55</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Work partitionning</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>Scope of the product</td>
<td>56</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Product boundaries</td>
<td>56</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Use case list</td>
<td>56</td>
</tr>
<tr>
<td>5.6</td>
<td>Usability requirements</td>
<td>60</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Ease of use</td>
<td>60</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Ease of learning</td>
<td>61</td>
</tr>
<tr>
<td>5.7</td>
<td>Non-functional requirements</td>
<td>61</td>
</tr>
<tr>
<td>5.7.1</td>
<td>User interactions</td>
<td>61</td>
</tr>
<tr>
<td>5.8</td>
<td>Conclusion</td>
<td>61</td>
</tr>
<tr>
<td>6</td>
<td>Implementation of the Editor</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction to GEF</td>
<td>62</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Draw2D</td>
<td>63</td>
</tr>
<tr>
<td>6.1.2</td>
<td>The GEF framework: MVC architecture</td>
<td>65</td>
</tr>
<tr>
<td>6.1.3</td>
<td>EditParts</td>
<td>65</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Requests</td>
<td>67</td>
</tr>
<tr>
<td>6.1.5</td>
<td>EditPolicies</td>
<td>67</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Commands</td>
<td>68</td>
</tr>
<tr>
<td>6.1.7</td>
<td>Reflecting model changes</td>
<td>69</td>
</tr>
<tr>
<td>6.1.8</td>
<td>Palette</td>
<td>69</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>6.2 Building the Binding Editor</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>6.2.1 Managing the model</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>6.2.2 Implemented use cases</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>6.2.3 Structure of the Editor</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>6.2.4 Conclusion</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>82</td>
<td></td>
</tr>
</tbody>
</table>
# List of Figures

2.1 Example of an XML document .............................................. 11  
2.2 Example of an XSL stylesheet .............................................. 14  
2.3 Linking XML document to XSL stylesheet .............................. 15  
2.4 First step of the XSL transformation ................................. 15  
2.5 Second step of the XSL transformation .............................. 16  
2.6 A small UML model representing an Author class .................. 17  
2.7 UML model of Fig.2.6 translated in XMI .............................. 18  
2.8 Screen capture of mediation configuration in TIBCO’s Business Studio ... 20  
2.9 Example of a job in Talend Open Studio .............................. 21  
2.10 Data warehouse architecture (OLAP and ETL) ....................... 22  
2.11 Example of a Star schema .............................................. 23  
2.12 Flow objects representations ........................................... 24  
2.13 Connecting objects representations ................................. 24  
2.14 Swimlanes representations ............................................. 24  
2.15 Artifacts representations ............................................. 24  
2.16 Metamodels for the ATL examples from Section 2.6 ............... 25  
2.17 Example of use of the ATLAS Model Weaver ....................... 28  
2.18 Using the AMW to declare ATL transformations ................... 28  
3.1 A simple EMF model and the corresponding generated source code . . . 32  
3.2 A simple EMF model and the corresponding generated source code . . . 33  
3.3 A schematized layer view of Wazaabi ................................ 34  
3.4 The four components of the Wazaabi framework .................... 35  
3.5 High level view of the structure of the binding metamodel in Wazaabi Binding 36  
3.6 Extension of the BPMN notations in the Wazaabi Weaving model ........ 37  
3.7 Ecore representation for context and location packages ............ 37  
3.8 Ecore model of the binding processes in Wazaabi .................. 38  
3.9 Translation of the Ecore model of Fig.3.8 as a UML class diagram ... 39  
3.10 Weaver plugins installed in runtime instance of Eclipse ............ 40  
3.11 Widgets in Wazaabi UI are all Binding Contexts .................. 42  
3.12 Binding model in Wazaabi UI ....................................... 43  
3.13 BPMN model of the Binding object from Wazaabi UI ............. 44  
4.1 Analogy between matched rules and Activities ..................... 48  
4.2 Modification of Fig.4.1 to remove the “laziness” of the second rule ... 49  
4.3 Simple BPMN representation of an ATL process .................... 49
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Extension of BPMN for representing “Changed” events and justification</td>
<td>51</td>
</tr>
<tr>
<td>4.5</td>
<td>Extension of BPMN for representing “Refresh” events and justification</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>The work context diagram for the binding editor</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>Use case diagram of the Binding Editor</td>
<td>56</td>
</tr>
<tr>
<td>6.1</td>
<td>The architecture of the Draw2D system</td>
<td>63</td>
</tr>
<tr>
<td>6.2</td>
<td>Hierarchical examples of Draw2D figures</td>
<td>64</td>
</tr>
<tr>
<td>6.3</td>
<td>The MVC architecture of the Graphical Editing Framework</td>
<td>66</td>
</tr>
<tr>
<td>6.4</td>
<td>Similar structure between EditParts and Figures</td>
<td>66</td>
</tr>
<tr>
<td>6.5</td>
<td>Extension of the Binding Model for the purpose of the editor</td>
<td>71</td>
</tr>
<tr>
<td>6.6</td>
<td>Partial class diagram of the Wazaabi Binding Editor</td>
<td>80</td>
</tr>
<tr>
<td>6.7</td>
<td>Typical binding process in the Wazaabi Binding editor</td>
<td>81</td>
</tr>
<tr>
<td>6.8</td>
<td>Binding Process from Fig.6.7 viewed with the EMF tree editor</td>
<td>81</td>
</tr>
</tbody>
</table>
Résultat

Ce mémoire de fin d'étude a pour but d'analyser et de contribuer à la sémantique de représentation de l'infrastructure, orientée transformation de modèle, qu'est le Wazaabi Weaver ainsi que de proposer un prototype d'éditeur graphique pour celui-ci. Ce travail s'inscrit dans le cadre du Model-Driven Architecture. Le MDA est un paradigme de conception de logiciels centré sur l'élaboration de modèles et la génération automatisée de code à partir desdits modèles. Afin de mener à bien les objectifs de ce mémoire, nous introduisons tout d'abord un panel varié de domaines de l'informatique qui font intervenir la transformation de données. Nous abordons, parmi d'autres, les processus ETL, le standard XSLT ou encore le langage de transformation ATL. Dans un deuxième temps nous présentons Wazaabi et le Weaver en détail ainsi que l'outil principal de modélisation sur lequel ils sont basés, à savoir EMF. L'étude approfondie du Weaver ainsi que la conscience de ce qui existe en matière de transformations nous permettent finalement d'analyser pertinemment la conception du Weaver, de porter un regard critique sur les éléments suggérés en matière de représentation des transformations ainsi que de contribuer à la sémantique de cette représentation et, enfin, de programmer un exemple d'outil graphique intégrant cette sémantique.
Acknowledgements

First and foremost, I would like to thank Mr. Esteban Zimányi for allowing me to carry out this master thesis in his department.

I would also like to thank my supervisor for this thesis, Sabri Skhiri, for his guidance, his patience and for keeping me in the right tracks, and this throughout the year.

I also wish to express my gratitude to Olivier Moïses for always finding some time in his busy schedule to help me along the way.

I am also grateful to Hervé Bath, Eric Delacroix and everyone at Euranova for welcoming me in their offices and providing me with propitious working environment.

I also would like to thank my family for their support and particularly my mother for reading and reviewing my work despite its complexity.

Finally, this work would not have been possible without the support of my lovely Aude who always provided me with all the help she could, who believed in me at all times and who gave me the strength to bring this thesis to fruition.
Chapter 1

Introduction

1.1 Background

MDA, which stands for Model-Driven Architecture, is a software design approach that was introduced by the Object Management Group (OMG). MDA is a subset of the greater category that is Model-Driven Development (MDD). The idea behind MDD is to use models as the primary element for the development process. A model is an abstraction that is semantically closer to the application domain than plain computation. A good model makes sense from the point of view of the user and allows for better communication amongst participants involved in a same project. Another interest of using models is, usually, to then use MDD tools to generate code from the model. Model-Driven Development has been around long before the OMG introduced MDA but never was a specification.

MDA is OMG’s initiative in the field of Model-Driven Development. It provides a set of guidelines for the specification of models and encompasses a set of modeling standards. Among those are such standards as the Unified Modeling Language (UML), the XML Metadata Interchange format (XMI) or the Meta-Object Facility (MOF). The Model-Driven Architecture paradigm is on the way of becoming a standard approach for building industry-grade software. The idea behind it is that by designing a model that describes the business logic, MDA tools can be used to interpret the model and generate the application.

Among the most important frameworks of MDA is EMF (Eclipse Modeling Framework) which is capable of generating a complete set of Java classes from a model and its specification described in XMI. On top of that, EMF also generates Eclipse plugins that allows to create, view and edit the instances of the designed model in the built-in EMF editor. This is rendered possible by the fact that EMF provides the model objects with very efficient notification mechanisms that allow for runtime synchronization between the instance of the model and its image in the editor. Model instances created with EMF can be persisted in XMI to be used with other MDA tools. EMF is an extremely powerful tool that is found at the core of numerous development tools, usually somewhat integrated into Eclipse.

Model-to-Model transformation (M2M) is a critical aspect of MDA. It is the process of transforming one model into a model of another type given a set of rules that bind the two. Because everything revolves around models in MDA, it is naturally of great interest to be able to generate models from a model.
1.2 Motivation

Despite the *de facto* standardization of the MDA approach, model transformation in application development is still mostly used to transform a platform-independent model (PIM) into a platform specific model (PSM). There is no tool that enables the conversion of a class diagram into an entity relationship diagram for instance. Alcatel-Lucent has been developing such a tool, the Wazaabi Weaver. This Weaver was developed for Alcatel-Lucent by Olivier Moïses in collaboration with Euranova. Their personal interest would be to be able to graphically generate a user interface (UI) by transforming the database model. Furthermore, the Wazaabi plugins have the very interesting particularity of using “live” models. This means that the user works on models at the same time as they are being run. The interest of live models will make more sense later in this work.

The aim of this work will be to analyse the work that was done on the Weaver to this day, to estimate the legitimacy of the choices that were made regarding the representation semantics, to propose improvements to said semantics should we deem it necessary and to design a graphical editor that implements said semantics.

1.3 Structure

This thesis will cover four aspects, typical of the scientific approach. We will start off by establishing a state of the art of existing technologies that involve data transformation. We will try to have an as broad view as possible, looking at technologies that relate to transformation in dissimilar ways and, for some of them, take a look at a particular software implementation. In a second part, we will study the issue at hand. We will examine the Wazaabi Weaver on which the rest of this work will be based. We will try to understand how it works in order to have a better understanding of the target implementation environment for our transformation semantics. In the third part, we will analyse the work that was done so far regarding the binding model. Based on what we will have learned throughout the previous two chapters, we will explain how the work on the Weaver attends to some issues in M2M. We will also analyse the work that was done so far regarding the representation semantics for model transformations based on the binding model. Finally, in the last part, we will formally specify the requirements for the Weaver’s graphical editor and we will develop a prototype.
Chapter 2

Transformation: a state of the art

Model transformation is the process of transforming a model $M_a$ conforming to a metamodel $MM_a$ into another model $M_b$ conforming to metamodel $MM_b$. It is a critical aspect of Model Driven Architecture (MDA). Several model transformation languages have been developed over the years, each of them with its own specificities:

- **ATL** [39]: a transformation language developed by the INRIA (Institut National de Recherche en Informatique et Automatique)
- **Beanbag** [45]: an operation-based language for establishing consistency over data incrementally
- **GReAT** [4]: a transformation language design for Model Integrated Computing
- **Kermeta** [25]: a general purpose modeling and programming language that also enables transformation
- **Lx family** [5]: a set of low-level transformation languages
- **MOLA** [17]: a graphical high-level transformation language built in upon Lx.
- **MT** [41]: a transformation language developed at King’s College, London
- **QVT** [26]: the OMG has defined a standard for model transformation called MOF/QVT or in short QVT.
- **SiTra** [1]: a pragmatic transformation approach based on using a standard programming language e.g. Java, C#
- **Stratego/XT** [42]: a transformation language based on rewriting with programmable strategies
- **Tefkat** [21]: a transformation language and a model transformation engine
- **VIATRA** [36]: a framework for transformation-based verification and validation environment
CHAPTER 2. TRANSFORMATION: A STATE OF THE ART

2.1 XML TRANSFORMATION

XSLT or XSL Transformation\(^1\) is a W3C recommendation. It is a declarative language that enables the transformation of XML documents into various other XML compliant output. The XML document is transformed using an XSLT stylesheet through a XSLT processor.

2.1.1 XML

XML stands for eXtended Markup Language. This W3C standard introduced in 1998 allows to store information through the use of tags, much like in HTML\(^2\). XML documents have a tree-like structure. XML is used to store and exchange information in an ever-growing amount of real life applications. The major advantage that XML holds over other information storage technologies is its universal nature. Moreover, its simple and intuitive structure make it a fairly easy to get into.

Fig.2.1 shows a simple example of an XML file\(^3\) An XML document is comprised of elements. An element is defined by an opening tag and a closing tag. Tags are words (or phrases) enclosed between < and >. Both the opening and closing tag share the same name.

---

\(^1\)XSL stands for XML Style Sheet

\(^2\)HTML is actually an XML-like structure. XHTML is the XML compliant “version” of HTML

\(^3\)Example drawn from the W3C XPath tutorial http://w3schools.com/xpath/default.asp
with the addition of a “/” before the name in the closing tag. In the example at Fig.2.1, 
bookstore, book, title, author, year et price are elements of the document, language 
is an attribute of the title element, ”english” is the value of the language attribute 
and Harry Potter is the content of the title element. Therefore, the collection of the 
book elements is the bookstore element’s content. The first line of this XML exemple is 
the XML declaration. It specifies the version of XML used (“1.0” or “1.1”) along with 
the character encoding. To be well-formed, an XML document must start with such a 
declaration.

Given XML’s tree-like structure, the “elements”, as defined above, are commonly ref-
fered to as “nodes”. When considering a given node in the tree, a parent node (respectively 
child node) will refer to a node placed higher (respectively lower) in the tree hierarchy. 
In other words, an element A that is part of the content of another element B is consid-
ered a child element of B, B being a parent of A. Here, <bookstore> is a parent node of 
<author> et <title> is a direct child node of <book>.

2.1.2 XSL

XSL (XML Stylesheet Language) is the language for expressing stylesheets for XML doc-
uments. XSL stylesheets are also XML documents as they conform to the XML speci-
fications. Given a XML document, an XSL stylesheet will be used to specify how this 
document should be presented on a presentation medium. The stylesheet along with the 
XML document are processed by an XSL processor to produce the desired output. Two 
steps are involved in the XSL processing. The first step is the tree transformation dur-
ing which the structure of the source tree is transformed (possibly significantly) to produce 
the result tree. The second step is the formatting which is enabled by including format-
ing semantics in the result tree. These formatting objects are interpreted by a formatter 
(typically a browser).

For that matter, XSL could be described as a family of recommendations for defining 
XML document transformation and presentation [28]. It consists of three W3 recommenda-
dations:

XSLT (XSL Transformation), a language for transforming XML documents into virtually 
anything,

XPath, a language used for navigating in an XML document,

XSL-FO (XSL Formatting Objects), an XML vocabulary for defining formatting seman-
tics. This one will not be investigated further as it merely specifies the layout of the 
document resulting from the transformation.

2.1.3 XPath

XPath is a language allowing to define access paths within XML documents. An XPath 
expression is composed of one or more steps separated by slashes (“/”). Each step is 
composed of an axis and a node test separated from each other by two colons (“::”). 
The node test can be followed by zero or more predicates enclosed in brackets (“[]”). An 
example of an XPath expression for our example at Fig.2.1.
CHAPTER 2. TRANSFORMATION: A STATE OF THE ART

2.1. XML TRANSFORMATION

Expression 2.1.1 Example of an XPath expression

descendant::bookstore/child::book/child::title[attribute::lang='english']

In this example child::title[attribute::lang='english'] is a step in which descendant is the axis, bookstore is the node test and [attribute::lang='english'] is a predicate that tests whether the “title” node has a “lang” attribute whose value equals “english”. XPath also supports a more commonly used abbreviated syntax. The above expression would then be written as follows:

Expression 2.1.2 Example of an abbreviated XPath expression

//bookstore/book/title[@lang='english']

This expression will result in the selection of any <bookstore> node, no matter its position within the context, having a direct <book> node child having a lang attribute that evaluates to english.

The axes in XPath determine directions that the path will follow with respect to the current node in the tree. Examples of axes include:

- **parent** which will fetch the parent node of the current one,
- **following** which will fetch all that follows the closing tag of the current node,
- **self** which is the current node.

For use in the predicates, XPath allows to execute certain operations on the elements of an XML document. To that purpose, it defines the operators for addition, subtraction, multiplication and division but also the comparison operators (=, !=, <, >=,...) and the logical operators **and** and **or**.

XPath is intensely used in XSLT as we will see in the next section.

2.1.4 XSLT

XSLT is the language used, as explained above, for the structural transformation of the source XML tree. XSLT elements are defined by using the namespace http://www.w3.org/1999/XSL/Transform. As mentioned earlier, an XSLT stylesheet should be an XML document. Doug Tidwell [40] explains that XSLT was designed according to several design goals, some of those being that:

- an XSLT stylesheet should be a well-formed XML document. Therefore an XSLT stylesheet could be transformed by another XSLT stylesheet.
- XSLT is based on pattern matching. An XSLT stylesheet consists of a set of rules (called templates). These rules are given an argument that, whenever matched in the source XML document, will “trigger” the execution of the rule.
XSLT differs from any other programming language in that a variable’s value cannot be changed once it has been initialized. This choice was made in order for several rules to be applied simultaneously without side effects. Indeed, if variables could be changed, one rule could modify its value, thus potentially altering the outcome of another subsequent rule.

An XSL transformation does not affect or alter the source XML document. The word transformation is a convention. Therefore, from this point forward, we will refer to an XML document as being “transformed” to refer to the aforementioned XSLT process of applying a stylesheet to the XML document in order to produce a desired output.

As we mentioned earlier, the core element of an XSLT stylesheet is the rule which is defined by the element `<xsl:template match='some_xpath_expression'>`. An example of an XSL style sheet is illustrated at Fig.2.2 In this example, the goal is to output some HTML code that will present the content of the source XML document in a certain way. The first rule of this stylesheet will be called whenever a root `bookstore` element is matched (maximum once, in this case, if the source XML document is valid). In this...
CHAPTER 2. TRANSFORMATION: A STATE OF THE ART  2.1. XML TRANSFORMATION

<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="Library.xsl"?>
<bookstore>
    <book>
        <title language="english">Harry Potter</title>
        <author>...
    </book>
</bookstore>

Figure 2.3: Linking XML document to XSL stylesheet

<html>
<body>
<h2>My Library</h2>
<table border="1">
<tr bgcolor="red">
<th>Title</th>
<th>Author</th>
</tr>
</table>
</body>
</html>

Figure 2.4: First step of the XSL transformation

rule, we can see a call to an iterative function (xsl:for-each) that will iterate on each book child of a bookstore element. For each of these elements, it will output the value (xsl:value-of) of its title and author child nodes in separate HTML table cells.

Given an XSL stylesheet, an XML document that should be “transformed” according to that stylesheet should be linked to it like illustrated at Fig.2.3 where our sample XML document from Fig.2.1 is linked to the stylesheet from Fig.2.2 (Library.xsl). Doing so will notify the XSLT processor that this XML document should be transformed according to the rules specified in Library.xsl. In this example, the reference to the stylesheet Library.xsl is a relative path which in this case implies that this XML document has to be in the same folder as Library.xsl. Absolute paths can be used instead.

When both the XML and corresponding XSLT documents are fed to an XSLT processor, the latter attempts to apply the rules of the stylesheet to the input XML document. The processor thus builds the tree of the document, positions itself at its root and looks for the best suited rule that matches this context. All stylesheets should have a base rule that matches the document’s root. This base rule will usually contain calls to other rules, and so on.

In this example, the root ("/") is matched by the first rule of our sample stylesheet and the processor starts executing the rule. The beginning of this rule just consists of writing text “as is” in the output document (Fig.2.4).

At this point, the processor encounters an XSL instruction (<xsl:for-each>, characterized by the xsl namespace)). This instruction specifies a loop that iterates on every book children of the top bookstore element. For each iteration of the instruction, the context node will be the current book element. Therefore, each time the loop executes, it
CHAPTER 2. TRANSFORMATION: A STATE OF THE ART

2.1. XML TRANSFORMATION

Figure 2.5: Second step of the XSL transformation

asks to apply the template rules that match the current book node (‘‘.’’). When the processor encounters such an <xsl:apply-templates> instruction, it will look for and apply the best-suited template rule that matches the type of node specified by the select= attribute. In our case, this rule is the second template on the stylesheet. This template, again, writes some HTML code that declare a table row (<tr>...</tr>) that contains two cells (<td>...</td>). This cells contain the result of the XSLT instruction <xsl:value-of .../> that returns the content of the node provided by the select= attribute. Because our sample document contains two book elements, the for-each loop will execute twice, once for each of those elements. Exiting the for-each loop, the processor executes the end of the main template rule that essentially consists of closing the previously opened HTML tags in the output document. The final output is shown on Fig.2.5

2.1.5 XMI

The XML Metadata Interchange format (XMI) is an OMG standard that allows to serialize models (and metamodels) as XML documents. The standards defines the structure (schema) of an XMI document as well as a namespace for XMI declarations. However, XMI is much more than a set of rules for serializing models in XML. It also specifies how to create XML schemas from models or how to reverse-engineer models from XML documents. Instead of diving into extensive explanations that go beyond the scope of this work, let us instead illustrate the use of XMI by means of an example.

Let us consider the UML diagram from Fig.2.6. It contains an Author class which has four attributes among which one (the gender attribute) that can only take the values
listed in the Gender enumeration. Fig.2.7 shows the same model, but serialized in XMI. As we can see, XMI does not define a vocabulary to represent model elements. Still, it does define some attributes that are independent of model, i.e., that apply no matter the model. Among those, we can see in this example the xmi:id, which is used to identify each element uniquely, and the xmi:type, which indicates the type of the element. In this example, the names of the tags all belong to the UML namespace because this document describes a UML model. But any model can be represented in XMI. It is the structure of the document that defines the structure of the model. Each tag corresponds to a model object and the containment hierarchy of the XML tags reflects the containments in the model.

2.1.6 Model transformation with XMI and XSLT

We have seen that XML stylesheets can be applied to XML documents in order to transform it following a set of rules. We have also seen that XMI specifies how to express a model as an XML document in a way that it can be interchanged and understood by processes that now of XMI. Therefore, it is worthy to note that model transformation can be achieved in pure XML by applying stylesheets to XMI documents. Of course this is not an ideal way to do it for several reasons. The sole fact of working with XMI models is unpractical. XMI should be a means for modelling tools to persist models. Furthermore, the process of writing XSLT stylesheets to transform XMI documents would be quite fastidious. We will nonetheless encounter a similar idea when we introduce ATL later in this chapter.

2.1.7 Altova XMLSpy™

XMLSpy is a very powerful and interesting tool for working with XML documents. It has numerous XML related features such as [3]:

- Intelligent XML editor;
- XSL / XSLT 1.0/2.0 editor, XSLT debugger & XSLT profiler;
- Graphical XML editing and text-based XML editing views;

Note that in the case of the gender attribute of the Author class, the value of the xmi:type attribute contains the value of the xmi:id attribute of the Gender enumeration. As for all the other attributes of the Author class, the uml namespace defines the primitive type but there is no such thing as uml:Integer. Therefore the href attribute specifies the exact type of that primitive type.
Figure 2.7: UML model of Fig.2.6 translated in XMI
CHAPTER 2. TRANSFORMATION: A STATE OF THE ART

2.2 XSLT ELSEWHERE

- Schema-aware XQuery editor, XQuery debugger & XQuery profiler;
- XML validation with advanced error handling;
- XPath auto-completion and XPath Analyzer window;
- Visual XML Schema editor;
- Database integration;
- And many more...

XMLSpy does not provide much graphical editing support for their XSLT editor as, according to Altova, most XSL developers prefer a text based XSL editor. Therefore, the process of declaring XSL Transformation in XMLSpy remains the same as with a simple text editor, if not for such common IDE features as auto-completion and helper windows.

XMLSpy incorporates two XSLT processors, depending on which version of XSLT is used (1.0 or 2.0). When an XML document is the “current document” in the XMLSpy window and if this document is linked to a stylesheet as we have seen earlier (Fig.2.3), a simple click on the “XSL Transformation” button ( ) will run the XSLT processor and display the output of the transformation. Furthermore, XMLSpy will automatically detect which of its two XSLT processors should be used depending on the version of XSLT that is declared to be used in the stylesheet (e.g., version 1.0 in the example at Fig.2.2).

2.2 XSLT elsewhere

Other software make use of the XSLT technology, though often in less obvious ways. An example of such software is TIBCO’s Business Studio, an environment based on the Eclipse RCP\(^5\) that allows to design and implement business process solutions, and more specifically TIBCO ActiveMatrix Service Bus. The ActiveMatrix Service Bus is a lightweight Enterprise Service Bus (ESB) incorporated in Business Studio that helps organizing mediation in an SOA\(^6\) infrastructure. XSL Transformation in TIBCO’s ActiveMatrix is used in mediation. Mediation in Enterprise Service Buses [9] is essentially the communication between the different services. In SOA, services communicate by exchanging messages in XML format. These messages need to be transformed to conform to each target services. That is what is meant by mediation in Enterprise Service Buses. A screen capture illustrates that concept in TIBCO’s Business Studio at Fig.2.8. We can see in this image that the message undergoes atomic operations on its way from the mediation interface to the target service’s interface. The second operation, which is a mapping, is detailed in the lower part of the screen. Fields of the original XML message are mapped to fields in the message structure expected by the target interface. These graphical mappings translate to XSLT processing in the bus.

\(^5\)Rich Client Platform

\(^6\)Service Oriented Architecture, in which ESBs are a key component
2.3 ETL

*Extract, Transform, Load* refers to a process used with databases and especially with *data warehousing*. ETL is not a language and does not specify a standard or a norm. It only describes a type of process by means of specifying the different actions it should realize and to what end. Ralph Kimball, a renowned visionary in the field of data warehousing gives a definition of an ETL system:

“A properly designed ETL system extracts data from the source systems, enforces data quality and consistency standards, conforms data so that separate sources can be used together, and finally delivers data in a presentation-ready format so that application developers can build applications and end users can make decisions.” [19]

ETL involves three phases which consist of, as its name clearly implies:

**Extracting** data from external sources which can be of various nature (relational databases, XML documents, CSV files, . . . ). The ETL process must be able to parse different files that may have very different structures and fetch the relevant information;

**Transforming** the extracted data according to the operational needs. This transformation can involve many operations such as filtering values, sorting them, fixing mistakes in the extracted data, rewriting values to conform to the end format (data cleansing), deriving calculated values, joining values from different sources, and many more;


**Loading** the data into the end target (a data warehouse in most cases [32]). The way this step is performed varies widely depending on the way the business requirements (e.g., some data warehouses overwrite some values while others will keep a history of past values).

Obviously, in the context of this work, the aspect of ETL that will be of interest to us is the transformation. Since there is no specification as to how transformation is to happen in the course of an ETL process, we will study one specific ETL tool, Talend Open Studio that is.

### 2.3.1 Talend Open Studio

Talend Open Studio is an open source framework base on the Eclipse platform. It allows graphical definition of ETL tasks called “jobs” to be compiled in either Perl or Java. The job is the very ETL process that we want to realize. In all generality, these jobs comprise three types of components: source, output and transformation components. Typically, the data will go from the source(s) to be modified accordingly by the transformation components to produce the desired output(s). In the graphical editor of Talend Open Studio, the job is modeled as a flow of the data through the components. This concept is illustrated on the (rather complex) example at Fig.2.9. We can see on this picture that,

![Figure 2.9: Example of a job in Talend Open Studio](image)

initially, some data is stored on a flat file called “Orders”. This data is organized in rows. Values are separated by semicolons\(^7\) and each new line is a new row. The content of the “Order” file will flow through a transformation component that will map these rows to

\(^7\)Default value in TOS. Any other separator can be specified in the component’s settings
either of the two output files depending on whether the order is deemed valid or not according to the configuration of the mapping component.

2.4 OLAP cubes

[hl] OLAP stands for *OnLine Analytical Processing* and describes an approach for querying multidimensional data. It is part of the much broader and fashionable concept known as Business Intelligence (BI). As we can see on the schema at Fig.2.10, both OLAP and the previously introduced concept of ETL play quite big roles in the usual data warehouse architectures. An OLAP system is based on the concept of the OLAP cube. This multidimensional view of the (meta)data is made possible by some specific relational database schemata (the *star schema* and *snowflake schema*) used particularly with data warehouses. In such a schema, the main data (the *facts*) is contained in the *fact table*. These facts are classified along several dimension. The data associated with these dimensions is stored in smaller tables called *dimension tables*. The tuples from the dimension tables are characterized by a simple primary key. These keys are used in the facts from the fact table to reference the appropriate tuples from the dimension tables. An example of such a schema is illustrated at Fig.2.11. The transformations involved in OLAP are a bit similar to those of ETL processes. The OLAP system first needs to fetch the data from the different tables (dimension and fact). It then has to manipulate these data (aggregation, rotation, slicing)
in order to produce the output requested by the client (graphs, tables, ...).

2.5 BPMN

BPMN, which stands for Business Process Modeling Notation, is a standard notation from the Object Management Group (OMG) for modeling Business Processes. It would seem at first that this is not related to transformation per se. However, a transformation is a process and can therefore be modeled graphically, as is done in Talend Studio for the declaration of ETL processes. BPMN defines a series of symbols that can be divided into four categories [43]:

Flow Objects are the core elements of a BPD (Business Process Diagram). There are three types of Flow Objects:

- **Events** are things that happen in the course of the modeled process (Fig.2.12a);
- **Activities** are work that is performed (Fig.2.12b);
- **Gateways** control the forking or merging of paths (Fig.2.12c).

Connecting Objects connect the Flow Objects together. Once again, there can be three types of connecting objects:

- **Sequence Flows** determine the sequence of the activities (Fig.2.13a);
- **Message Flows** represent a message that is passed from one Business Participant to another (Fig.2.13b);
- **Associations** are used to associate Artifacts (see below) to Flow Objects (Fig.2.13c).

Swimlanes are a concept found in many process modeling methods. It is used to organize activities into separate categories in such way to illustrate different functional entities. Two different types of swimlanes are distinguished in BPMN:
**Pools** represent participants in the process. They are graphical containers for activities (Fig. 2.14a);

**Lanes** partition Pools. They provide additional flexibility for the partitioning and organizing of activities (Fig. 2.14b).

**Artifacts** allow some flexibility in a BPD design. They are used to give additional information to the Business Process Diagram. Such Artifacts are such as those listed below but modelers are free to invent their own Artifacts:

- **Data Objects** are associated (through the use of Associations as defined above) to activities to represent how data is required or produced by said Activity (Fig. 2.15a);
- **Groups** are merely used for documentation. They do not affect the modeling *per se* (Fig. 2.15b);
- **Annotations** are a mechanism used to provide additional text information to the diagram (Fig. 2.15c).

BPMN can be used to model Business to Business (B2B) processes, which depict interactions between business entities, as well as to model Internal Business processes, which focus on an internal process, involving activities that are not visible to the public.

2.6 Model Transformation: ATL

As we have seen earlier, there already exists a significant amount of transformation languages that can apply to model transformation. However, a lot of them are quite domain
specific and they do not apply to “live” models which is the very particularity and interest of Wazaabi. Because there are quite a lot of them, we will only get into one: the ATLAS Transformation Language. ATL is a model transformation language developed by the INRIA in answer to the Object Management Group (OMG) request for proposal for a model transformation language compatible with QVT\textsuperscript{8}, MOF\textsuperscript{9} and other MDA recommendations. ATL was one of the submitted responses to that RFP and is the most advanced implementation of QVT with a large community around it. Moreover, it has been integrated as part of the Eclipse Modeling Project (EMP).

ATL is a “hybrid” language in that it allows both imperative and declarative programming ([39]). Although it is recommended to use a declarative approach as much as possible, imperative constructs are allowed as some situations may be too complex to express in a declarative way.

An ATL program has to be given both the source and target metamodels (the descriptions of the source and target models). Model-to-model transformation operations are called modules in ATL. Model elements can also be transformed to primitive data types. Such operations are called queries. We will try to illustrate the basic concepts of ATL through a series of simple examples based on the two metamodels given at Fig.2.16.

The core of an ATL transformation are rules that match specific elements from the source model and define how that element is to be transformed in an element from the target model. Below is an example of an ATL rule that matches an Author to a person:

```plaintext
rule Author {
    from
    a : MMAuthor!Author
    to
    p : MMPerson!Person (name <- a.name,
              surname <- a.surname
    )
}
```

In this example, the rule, called “Author”, will be applied each time an Author element is encountered in the source model. It will then create an element Person in the target model.

\begin{figure}
\centering
\begin{subfigure}{0.4\textwidth}
\includegraphics[width=\textwidth]{author_metamodel.png}
\caption{The Author metamodel}
\end{subfigure}\hfill
\begin{subfigure}{0.4\textwidth}
\includegraphics[width=\textwidth]{person_metamodel.png}
\caption{The Person metamodel}
\end{subfigure}
\caption{Metamodels for the ATL examples from Section 2.6 [39]}
\end{figure}

---

\textsuperscript{8}Queries/Views/Transformations is a model transformation standard from the OMG

\textsuperscript{9}The Meta-Object Facility is an OMG standard that originated from a need for a metamodel to define UML
and map the Person's name and surname to the Author's name and surname respectively.

In some cases, additional tests are needed in order to more precisely match source model elements. ATL provides something called “helpers”. Helpers in ATL are analogous to functions in C++: they are bits of factorized code that can be called anywhere in the ATL code. An example of a helper in our case would look something like this:

```plaintext
def: isMale() : Boolean =
  if self.gender = \#Male then
    true
  else
    false
  endif;
```

In this example, the helper entitled isMale() performs a boolean test on an Author element that checks if its gender attribute equals to “Male”. Our “Author” rule could be modified to make use of our new helper as follows:

```plaintext
rule Author {
  from
    a : MMAuthor!Author (s.isMale())
  to
    p : MMPerson!Person {
      name <- a.name,
      surname <- a.surname
      gender <- \#Male
    }
}
```

The rule now tests whether or not the matched Author element is a male or not. If it is, it will assign the enumeration literal Male to the gender attribute of the target Person element. Of course, this rule is definitely not an optimal way to go by but it serves its purpose of illustrating the use of a helper.

ATL rules can exist in two slightly different forms: called rules and lazy rules. A called rule can be seen as a special type of helper. It also is a factorized bit of ATL code that as to be called from an imperative section of code in order to execute but, as opposed to helpers, called rules can create target model elements. Lazy rules are just like called rules in that they share the exact same function and syntax but can only be called by other rules.

There is a bit more to ATL than what is presented in this section. However, this introduction gives us enough of a grasp of the semantics behind ATL programming. An interesting thing to note is that the semantics of ATL have quite a lot in common with XSLT: a source document is parsed until an element is found that is matched by a transformation rule which immediately executes a piece of code that uses parts or the whole of the matched source element in order to create a piece of the target document.

---

10Example might contain slight syntactical mistakes
2.6.1 Transformation directionality

In ATL, transformations are unidirectional [2]. The transformations operate only on read-only source models. During execution of a transformation, the source model can be navigated but no changes can be made to it. In order to create a bidirectional transformation in ATL, two transformations need to be implemented, the source model of the first becoming the target model of the second.

2.6.2 Model synchronisation

Model synchronisation is a very important concern in MDA and one that is not addressed by ATL [46]. Let us consider two models, a source and a target. The two models conform to different metamodels and the target model was obtained by a transformation written in ATL that took the source model as an input. After the transformation, both models can evolve independently. The two models can be used by different parties and modifications can be made to them independently. At that point the models no longer correspond with regards to the ATL transformation rules and the two models need to be resynchronised. Applying the ATL transformation will not suffice. The source model changes will indeed be propagated to the target model but, unfortunately, all changes that were made to the target model will overwritten and lost forever. The problem here is that ATL does not support incremental transformation. Incremental transformation implies that, instead of having a batch transformation re-creating the whole target model based on the source, only relevant transformations related to a specific model feature are executed whenever said model feature is altered.

2.6.3 Atlas Model Weaver

The Atlas Model Weaver (AMW) [11] is an Eclipse plugin that allows to create an EMF-based model (called the Weaving Model) that specifies a mapping between two models. The AMW has a generic purpose and can be extended to adapt to more specific business needs. The basic idea is this: a weaving model (WM) specifies relations between two metamodels. The weaving model itself conforms to a weaving metamodel (WMM). An example of generic weaving in Eclipse using the AMW is displayed at Fig.2.17. This weaving model specify relational mappings between two metamodels that present information about books. We can see for instance that an element of this weaving model specifies a relation of equality (Equals identification on the figure, where Equals is the type of relation and identification is the name given here to that relation) between the ISBN feature from the left-side model and the BookID feature from the right-side model.

Among the aforementioned extensions to that generic AMW Eclipse plugin is one that allows to graphically create simple ATL transformations using the ATLAS Model Weaver (as shown on Fig.2.18). In this example, we can see a rule called Member2Male that takes as input the Member element from the Families metamodel and outputs to the Male element of the Persons metamodel. This output consists of a binding between the fullName feature of the target element and the concatenation of the features firstName and lastName from the source element.

The AMW is therefore a layer that can be added on top of the ATL architecture and that provides semantics and graphical facilities to create ATL transformation by abstract-
ing from the ATL code. This tool however does not address the problem of incremental transformation. In order to execute the transformations declared by the weaving model, that model is transformed into an ATL model which is itself transformed into concrete ATL syntax. The AMW provides graphical capabilities to the creation of transformation but does not enhance the capabilities of the transformation engine.

Figure 2.17: Example of use of the ATLAS Model Weaver [11]

Figure 2.18: Using the AMW to declare ATL transformations [37]
2.7 Conclusion

Over the past years, several types of transformation languages have been developed but often, it seems, in a specific context and with a specific purpose. While the semantics that have been introduced in this chapter remain far from what we want to achieve with this work, lessons can be learned from the study of these technologies in order to efficiently design a transformation semantic that will apply to model-to-model (M2M) transformation. First of all, it seems the Business Process Modeling Notation is quite well suited for modeling a transformation process, somewhat as done for ETL processes in Talend Open Studio. We can also draw a conclusion from the comparison of XSLT and ATL: although those languages apply to completely different contexts, the fact that they have a similar semantics (i.e., pattern matching) tends to suggest that this way of proceeding is well suited for declaring transformation processes.
Chapter 3

The Wazaabi Weaving model

After having taken a broad tour of how transformation is solicited in various aspects of computing and already established technologies, it is time to study in depth the environment on which our work is based, Wazaabi that is. Wazaabi is both a program (an Eclipse extension to be precise) and a technology as we will further explain in this chapter. Wazaabi is consistently based on the Eclipse Modeling Framework (EMF) and a basic understanding of its concepts is required for the potential Wazaabi end-user. We will therefore try to introduce, quickly but extensively, the concepts behind EMF before properly diving into Wazaabi.

3.1 Introduction to EMF

EMF, for Eclipse Modeling Framework, is the cornerstone of the MDA paradigm. It is a modeling framework and code generation facility that is present in virtually every aspects of Model Driven Development (MDD). EMF is a very powerful tool that is worth being aware of, to say the least.

EMF defines a model to create models. Such a model is called a metamodel (a model of a model). In other words, EMF holds at its core a meta-metamodel that defines types such as EObject (supertype from which every EMF types inherit, much like the Object class in Java), a type EClass, a type EAttribute, and so on. All the terms in EMF are preceded by an “E” (or sometimes a lower case “e”) to signify that these are EMF’s definitions of those otherwise widely used concepts.

As we mentioned earlier in the introduction chapter, the idea behind model driven development is to base the building of an application on one or several models. This quite accurately describe the role of EMF. EMF facilitates the building of tools and applications based on a structured data model [38]. That structured data model is also referred to, in official documentation, as an EMF model or an Ecore model. EMF provides several ways to create an Ecore model. It can be generated from an imported UML diagram or XSD (XML Schema) file; it can be created with annotated Java code (through the use of the @model tag) or it can be built from scratch using the Ecore Sample Editor in Eclipse.
3.1.1 Ecore Model

Whether one chooses to create it manually or to generate it from an imported file, the creation of the Ecore model is the first and most critical step. The Ecore model is EMF’s representation of the model. The model in Ecore has a tree structure. The root of the diagram represents the project and its only child is the main package. Children of an Ecore package EPackage can be classes (EClass), data types (EDataType), annotations (EAnnotation), enumerations (EEnum), or subpackages (EPackage). Much in the same way, children of EClasses can be EAttributes, EReferences, EOperations and so on, children of an EEnum are the values it can take, and so on. Again, people who are familiar with UML or simply Java programming, amongst other technologies, will immediately recognize the vast majority of these terms, if not all. And they would be right for they mostly have the same meaning, the same semantics as in those technologies. These terms are, as we said before, appended with an “E” to signify that these represent concrete EMF elements.

Each and every node of the diagram (except for the root) possesses a number of properties that can be viewed and edited in the Properties panel in Eclipse. For an EClass, such properties are, among others: its name, whether or not it is abstract or a default value if required. For an EAttribute, some of those properties are: its type (EString, EFloat, EJavaClass,T,...), a lower bound and upper bound (to specify lists), whether or not it is unique, and so forth. EReferences can be viewed as EAttributes whose types are not primitive EMF types but, instead, reference other elements of the model. They have similar properties as EAttributes but with an additional important one, and that is the boolean “Containment” property. This property specifies whether the association is a composition or not. This property of a class which might not be trivial for everyone to implement manually in a Java class is a matter of setting a field to either “true” or “false” in EMF.

3.1.2 Generator Model

As soon as the Ecore Model is created, a Generator Model is generated along with it. Those two models look very similar but they serve a different purpose. Much like the Ecore model, nodes of the Generator Model possess a number of properties but these are divided into three sections: Ecore, Edit and Model. Ecore properties regroup all the properties of the corresponding selected node in the Ecore diagram but cannot be edited here. The two other sets of properties are used to configure the code generation. To summarize, the .ecore model is the proper model and changes to the model should be made on this one whereas the .genmodel is used to generate the various bits of code.

3.1.3 Code generation

At any point, code can be generated from the generator model. Code can be generated for any given sub-tree of the diagram but it is usually recommended to generate the whole project which produces an equivalent result. This way of proceeding helps to avoid any inconsistencies in the resulting code. The generated code is divided into two main categories: the model and the edit/editor code

1It is also possible to generate test code which is a complete suite of unit tests.
CHAPTER 3. THE WAZAABI WEAVING MODEL

3.1. INTRODUCTION TO EMF

(a) An example EMF model

(b) Generated source code

those plugins are installed in an Eclipse application. The model code is the translation in
Java of the EMF model.

The structure of the source code for the model correspond to the structure defined
in the Ecore model: to each EPackage defined in the model corresponds a Java package
in the source code. For each modeled object, an interface with the same name is generated
along with a concrete implementation of that interface. For example, for an EMF
model as simple as the one displayed at Fig.3.1a, the generated source code is as shown
at Fig.3.1b. Both model elements MyObject and MyOtherObject from the mypackage
EPackage have their corresponding interfaces, MyObject.java and MyOtherObject.java,
in the mypackage Java package. All the code generated as part of a model’s package will
be in the form of interfaces. The concrete classes implementing these interfaces, among
which in this case MyObjectImpl and MyOtherObjectImpl, are located in the subpackage
impl (here mypackage.impl). This way of proceeding enables the use of multiple inher-
itance in the EMF model. Indeed, in Java, a class cannot extend more than one class
but an interface can extend several interfaces. Therefore, if we were to modify our EMF
model from Fig.3.1a to add a new model object MyInheritingObject that extends the
two previous ones (see Fig.3.2a), the code generation (Fig.3.2b) would result in an inter-
face MyInheritingObjectImpl.java that extends the interfaces of the two other model
objects:

public interface MyInheritingObject extends MyObject, MyOtherObject {}

and a concrete class MyInheritingObjectImpl that implements this interface:

public class MyInheritingObjectImpl extends MyObjectImpl implements MyInheritingObject

The generated interface of a model object contains accessors (getters and setters) for
each of the object’s feature declared in the EMF model. These methods are off course
implemented by the concrete classes among other additional methods useful to EMF.
However, the implementations of these accessors, specifically the setters, have an important
particularity in EMF. Every “set” method in EMF includes a notification mechanism. The
concept of notifications will be explained in more details in Section 3.5.
CHAPTER 3. THE WAZAABI WEAVING MODEL

3.2. WAZAABI

"Wazaabi is a set of Eclipse plugins allowing the use of EMF based models for building parts of an application GUI" [22]

Wazaabi allows to graphically declare UIs (User Interfaces). The difference between Wazaabi and traditional MDA approach (which consists of generating code based on a model) is that, in Wazaabi, the declared UI is a “live” model. That means that the representation of the UI that is handled in the Wazaabi editor is none other than the UI itself. The power behind that concept is that all the “code generation” phase of the usual development process is avoided. The same code is used at the design and at runtime. Wazaabi makes good use of the adapter design pattern which ensures that any change made on the model side is immediately reflected on the UI. A simplified view of the concept behind Wazaabi is illustrated at Fig.3.3. This image presents Wazaabi as being composed of three layers. The base layer contains an EMF Graphical UI model. The layer above it contains the engines capable of rendering the model in the top presentation layer\(^2\). Each specific implementation of a GUI possesses its own engine. The top layer is the presentation layer itself where the UI components are rendered by the engines.

\(^2\)The hierarchy of layer expressed here refers to the usual layer hierarchy in IT (where each layer “is aware” of the ones below it) and is not to be mistaken with the vertical order in which the layers are represented on the the image.
The framework is actually divided into four distinct entities as illustrated on Fig.3.4. The models are, as explained earlier, the conceptual representations of the UI target technologies. They carry all the needed information to properly render a component in its target UI. The editor is that provided by EMF with a few customizations. It is to be totally replaced by the Architect on the long term. The Architect is a WYSIWYG\textsuperscript{3} GUI editor. Finally the engines are responsible for rendering a UI model in the presentation layer.

Since the Wazaabi project was started, it has evolved into a more complex, more ambitious project. The idea was to extend the mechanics implemented so far and explained above in order to do proper model-to-model transformation. This project became a separate project from Wazaabi while keeping all the technical aspects that makes Wazaabi what it is. Therefore from this point forward, we will differentiate the existing, published version of Wazaabi and the \textit{under construction} M2M project by referring to them respectively as “Wazaabi UI” and the “Wazaabi Weaver” as it is referred to by the involved parties. Both these projects are called Wazaabi because that word also represents the layered architecture (EMF models and engines that make the models “live”) along with the key guide line that \textit{“in Wazaabi, everything is a model”} \cite{22}. We will therefore, at times, use “Wazaabi” to refer to those shared ideas.

\section*{3.3 Binding model}

Originally the idea behind the work on binding in Wazaabi was to be able to bind several business models with several UI models. It is in that context that the work on binding in Wazaabi came to life. It started as a sub-project inside Wazaabi but quickly became a project of its own when it was realized that binding two models in the context of Wazaabi

\textsuperscript{3}What You See Is What You Get
at the time (binding UI models to business models, that is) could be extended to the M2M domain (Model-To-Model).

The Wazaabi Binding Model (Wazaabi Weaver) that we will study here was started as a project for the telecommunication company Alcatel-Lucent. To this day it has not been completed, or rendered public for that matter. Along with the source code for that project come two chapters of an under-development documentation [23]. It is to be noted that the said documentation marks a slight evolution compared to what had been implemented in the source code. The basis for what follows is the documentation since it defines what the binding model is intended to be.

In the Wazaabi Weaver, binding processes are defined by a metamodel, the binding-core.ecore metamodel. This metamodel consists of various packages as illustrated at Fig.3.5.

3.3.1 Location

The location package, detailed at Fig.3.7a provides mechanisms for specifying paths in an EMF model. There are three structural parts that EMF models consist of: EObjects, EReference and EAttributes. In a typical EMF tree, the EObjects will be the non-leaves...
nodes of the tree, the EAttributes will be the leaves of the tree and the EReferences will be the branches that link an EObject node to one of its children EObject nodes. If we take as an example the actual location metamodel, we can see that a model instance of that metamodel will probably contain a “Step” EObject which will have a “nameTest” EAttribute (being a leaf of the tree) and some “predicates” EReferences that refer to EObjects of type “Expression”.

The location package aims to provide an “XPath-like” way of referencing structural features of an EMF model. Indeed, as shown on the Ecore model at Fig.3.7a, a LocationPath is composed of a series of Steps, each of those comprising an Axis a nameTest and a series of predicates just like the XPath syntax.

3.3.2 Pointers

The pointer package defines pointers that allow to refer to runtime instances of a node based on two parameters: a context EObject and the name of a structural feature of that EObject whether it be an EReference or EAttribute.

Those pointers can also be evaluated in order to retrieve the value pointed by the pointer. If the pointer points to an EReference then the evaluation will return the list of the EObject refered to by that EReference. If however it points to an EAttribute then the evaluation will return the value of the said EAttribute.

EPointers are used by the binding engine and will not be instantiated or manipulated by the user.

3.3.3 Process

Binding processes are the true execution part of the binding model. They define the transformations between the models. The range of action of a binding process can go from simply changing the value of an attribute to completely changing the structure of the bound model. The concept and nomenclature for binding processes is directly borrowed from the OMG’s Business Process Modeling Notation (BPMN) which was introduced in the previous chapter. We therefore recognize the concepts of “events”, “activities”, “sequence flows”. Elements of binding process can be given parameters which are always of type LocationPath. These parameters are closely linked to the semantic role of the element to which they are attached.
CHAPTER 3. THE WAZAABI WEAVING MODEL

3.3. BINDING MODEL

Figure 3.6: Extension of the BPMN notations in the Wazaabi Weaving model [23]

The graphical definitions of events and activities in BPMN are extended in the Wazaabi Weaver to reflect their behavior.

Events In BPMN, events are something that happen in the course of the process and are represented by circles. In the Wazaabi Weaver, they have the same meaning but their representation is enhanced to include event parameters as illustrated in Fig.3.6a.

Activities In BPMN, activities are work to be accomplished when come across in the course of the process. Again their representation is here extended to include the parameters as illustrated in Fig.3.6b.

This metamodel is essentially that of a business process. That model is presented in its original EMF at Fig.3.8 while Fig.3.9 presents the same model as an UML class diagram. The base element in this model is an abstract element called BindingElement since it has no parent. Three elements then extends this base element: “Node”, “Edge”, which are both abstract, and “Binding Process”. Intuitively, nodes form the components of the process while edges link them together. Therefore, a node is characterized by incoming and
CHAPTER 3. THE WAZAABI WEAVING MODEL

3.3. BINDING MODEL

Figure 3.8: Ecore model of the binding processes in Wazaabi
3.3. Binding Model

The Binding Model is a part of the Wazaabi weaving model that specifies what action to take upon a certain occurrence of a certain event. In other words, the binding process’ activities define the “what” of the transformation and the events describe the “when”. What still needs to be defined is the “where” and that question is answered by the Binding Context. The concept of the context as in “binding context” in the Weaver is not to be confused with the concept of context in XPath. To avoid any confusion, the concept of contexts in the Weaver will always be referenced as a binding context in the rest of this work. In XPath, the context represent the node that is being evaluated; the node to which a certain path is applied. In the Wazaabi Weaver, the context represents the “use case”, so to speak, for a certain transformation. In other words, it describes a situation in which a certain (set of) binding process(es) should be executed along with the resource needed for that.

Location paths are applied to contexts which are essentially other locations in models. In order to “physically” locate the resources, binding contexts use element locators to perform the aforementioned task. Hence the binding context is composed of three types of elements: a set of binding processes, a set of element locators that refer to specific locations and a set of bindings that link element locators to parameters of the binding.
3.4 Using the Binding Model

The binding model can be used as any other EMF based plugin. An easy way to do so is to launch a runtime Eclipse application from a first Eclipse instance that contains the models and engines projects. In addition to all the plugins installed in the base Eclipse installation, the runtime instance contains builds off all the plugin projects located in the first instance’s workspace. The presence of those plugins in the runtime Eclipse application can be verified by checking the installation details in “Help → About Eclipse → Installation Details”. The plugin tab of the following window contains similar entries to those shown at Fig.3.10.

When in the runtime instance of Eclipse which contains the Wazaabi Weaver plugins, it is possible to create an instance of that binding model by creating a new “Binding Model” EMF Wizard project. The instance of the binding model is then created as with any EMF model editor.

Binding processes were explained in details in the previous section. We saw that in a binding process, events and activities have parameters that are all location paths. For instance, a “changed” event’s parameter will be the location of the model feature where it is to look for a change; a “copy” activity’s parameter will be the “target” location path.
where it is to copy the value located at its “source” parameter location path.

However, location paths are not absolute paths. They are applied to a context node. That is precisely where the element locators are of use. Location paths are put in relation with element locators through the “binding” elements. As previously stated, element locators are used to locate physical resource. Thus, a location path is evaluated on the “context” referenced by the locator bound together with it. The use of element locators rather than specifying full paths directly in the process’ parameters is relevant. It allows to define binding processes and elements to which they apply regardless of the exact locations of the metamodels on the host computer. Element locators are only evaluated during the execution.

The way to bind two models is thus as follows. A top level element BindingContext is created. As stated earlier, the binding context has children of three types: binding processes, element locators and bindings. The binding processes describe the transformation, the element locators reference physical resource by way of a \texttt{uri} attribute and the bindings link parameters of the binding processes to element locators. Let it be noted that bindings only bind Locators and Parameters that belong to the same parent BindingContext.

\section*{3.5 Transformation triggering}

A process is triggered when its starting event “occurs”. In other words, if for instance the first node of a binding process (which will virtually always be an event) is the \textit{Changed} event and if the value of the feature located at this event’s observable parameter is changed, then the event will have occurred and the process, which will therefore have been triggered, will flow into the next node as defined by the outgoing sequence flow.

This is rendered possible because EMF natively provides a mechanism of “notifications”. In EMF, every single modeled object is a \texttt{notifier} in that they all implement the \texttt{Notifier} interface. EMF also defines another type of objects called \texttt{adapters} which essentially are receivers of notifications. The Wazaabi Weaver Binding engine (and all other Wazaabi engines for that matter) makes good use of that mechanism. Whenever a model object’s feature is changed, it is done by a call to the appropriate setter method (i.e. \texttt{Object.setFeature(newValue)}). As mentioned earlier in the introduction to EMF, these setters include a mechanism that generates a notification. This essentially results in the call of the method \texttt{notifyChanged()} on any adapter elements that are associated with that notifier.

In the Weaver, these adapter classes are located within an \texttt{adapter} package of the engine. Each object of the model has its adapter in the package. Whenever a new model object needs to be created, the engine of the Weaver creates a new adapter for this type. In other words, when a new \texttt{ChangedEvent event} is instantiated, the engine will create a new \texttt{ChangedEventAdapter adapter}. During the same execution process, the \texttt{ChangedEvent} will be registered with the \texttt{ChangedEventAdapter} (which is a model object and, therefore, a \texttt{notifier}) by calling the method \texttt{event.eAdapters().add(adapter)}. But in the case of the \texttt{ChangedEvent}, the adapter also needs to receive the notifications sent by whatever model object it needs to “listen” to. Therefore, whenever the “observable” parameter of a \texttt{ChangedEvent} is set, the model object will notify its adapter that one of its features was change. The adapter will in turn notify the engine that its “observable” parameter was
set. Finally, the engine will fetch the object to whom belongs the “observed” attribute and add the adapter to said object’s list of adapters. Therefore whenever the “observed” value is changed in the model object, the object will send out a notification message that the ChangedEventAdapter will receive because it has been registered to that very emitter. The ChangedEventAdapter will just have check whether or not the notifications concerns the “observed” parameter and, if it evaluates to true, then the ChangedEvent has occurred, the corresponding transformation is triggered and the process flows into the next element.

3.6 Binding in Wazaabi UI

Since late 2009, the work on the Wazaabi Weaver as been put to stop. The concepts and mechanisms developed in the Weaver were transported and adapted to be integrated into Wazaabi UI. Here we will take a look at the evolution of the binding in Wazaabi (UI) and how the concepts of the Weaver were adapted and integrated into it.

In Wazaabi everything is a model. The core model of Wazaabi is illustrated at Fig.3.12a where we can see the different packages that compose this model. Most of those packages are involved in the modeling of graphical user interfaces (e.g., the widgets packages provides a generic metamodel for GUIs and the layouts package models the layout mechanisms for the widgets). The package that is of interest to us is the binding package which is expanded on Fig.3.12a. The main element of that package is the BindingContext which we recognize from our study of the Weaver. This EClass is extended by the Widget EClass from the widgets package (see Fig.3.11); the Widgets EClass being the parent class of any graphical element. A binding context, and therefore a widget, can have a series of binding properties. A binding binds a source to a target and is set off by one or more triggers. Triggers, much like in the Weaver, are events that will cause bound values to be updated accordingly. But in this “diluted” version of the binding model, available triggers (or events) have been exhaustively defined and can either be UI events such as the loss of focus on a widget or they can be the change of property in a model. The binding model has clearly been extremely simplified compared to that of the Wazaabi Weaver. The main differences between the two are as follows:

- The amount of activities available has been reduced to strictly one: the “copy”
CHAPTER 3. THE WAZAABI WEAVING MODEL 3.6. BINDING IN WAZAABI UI

(a) Binding in Wazaabi UI

(b) Example of use of the binding

Figure 3.12: Screenshots of both the binding model in Wazaabi UI and an example of its use

activity. When a binding is triggered, its sole purpose is to fetch a value at a source location and copy it at a target location.

• The whole concept of binding processes has been replaced by the sole Binding element. This element binds a “source” to a “target” and can be triggered by several events. Such a binding could be modeled by a binding process as defined in the Weaver which is what is represented at Fig.3.13

• The location mechanism does not have its own model package anymore. Instead, location paths are entered as strings of characters in the “source” and “target” attributes of a Binding element. The same syntax is used as in the Wazaabi Weaver but those strings will be parsed at runtime by engine to fetch the appropriate structural feature.

The purpose of the binding in Wazaabi UI is therefore to offer the possibility binding attributes from an interface to attributes from a business model. Typically those attributes will be numbers or text elements. An example of use would be to automatically update some text value in a business model according to a text input field in the user interface whenever the focus is removed from said text field. This is more or less what is done in the example of Fig.3.12b, where the name attribute of the root of a model called “library” is bound several times to different text values of the displayed UI model. For some of those binding, the triggering event is when that name attribute is modified (PropertyChanged).
3.7 Conclusion

The Wazaabi Weaving model provides a good infrastructure for creating binding processes. The model provides functionalities to reference parts of models, to link parameters from processes’ nodes to structural features of EMF models and to connect pieces together in order to build “live” binding processes thanks to EMF’s notification mechanism. Moreover by separating the physical location of the models from the binding process, the creation of binding processes is independent of the host system.
Chapter 4

Analysis of the binding model

4.1 Incremental transformation

An important issue of model transformation is how to apply the changes made on source model elements to target model elements. The simplest solution is to just run the transformation whenever we want to “commit” changes made to the source model. This approach is of course not optimal. When working with very big models, rerunning the whole transformation for a single modified text field will waste a lot of unnecessary execution time.

A more effective solution is to transform only the modified source model elements and reflect only them in the target model. Formally, a transformation is called incremental if individual changes in a source model lead to execution of only those rules which match the modified elements.

As mentioned previously, ATL does not currently support incremental processing. Each time a transformation is launched, the source model is fully read and the target model is completely rebuilt. Also if the transformation is launched again, the changes made to the target model by the user will be lost.

The Wazaabi binding model, however, allows for incremental transformation. As mentioned in a previous chapter, Wazaabi provides a mechanism that listens for structural changes in models. If a Changed event is declared and configured to “watch” a particular structural feature, then any change to that structural feature will trigger that event and, therefore, the binding process associated with it.

By defining a large amount of “small scale” transformation processes, incremental transformation can be achieved as described above. Indeed, each time a “listened to” change is made to a model, only the binding processes related to that modified structural feature will be triggered and hence, only the structural features targeted by those binding processes will be updated. In other words, each time a single change is made to the source model, the target model will be updated accordingly (if necessary), in real-time, and therefore be kept “in sync” at all time during the execution.

Of course, the relevance of the incremental transformation enabled by Wazaabi is dependent on the way the user defines the binding. If the user defines a single binding process which is triggered by any change in the source model and which defines the whole transformation from the source metamodel to the target metamodel; then the use of Wazaabi makes no sense. Indeed, in that case, each time the slightest change is made to it, the
CHAPTER 4. ANALYSIS OF THE BINDING MODEL

4.2 BUILDING ON WAZAABI

whole source model will be processed and transformed, which of course is exactly the opposite of what incremental transformation is about. The user would therefore be better off using straightforward ATL transformation which would be much easier to implement and would do an as efficient job and in a lesser time because of the higher complexity that Wazaabi has over ATL.

Another factor that affects the efficiency of incremental transformation is if we were to choose to use another transformation language such as ATL on top of the binding model\(^1\). In that case, the ATL rules that one chooses to associate to one’s activities would need to be well-defined and as specific as possible.

4.2 Building on Wazaabi

The Wazaabi Binding model does not define actual transformations but rather a framework to implement them. The binding model describes what kind of elements compose a transformation and how these elements can be pieced together to form transformations as well as providing the mechanism that handles the execution.

A transformation process in Wazaabi is essentially characterized by two elements: the events and the activities. The events define the where and the activities the what of the transformation. It is therefore the activities that describe exactly the manipulations and operations performed on the models. The binding model defines an abstract Activity element which has an undefined execute method.

In the last days of its development, the binding model was added several predefined activities. The idea was to implement a set of basic activities which, combined together, allowed to create every transformation process. In its current state, the binding model also defines four concrete activities, but only the first two are actually implemented: add, set, remove and replace. Intuitively:

- the add activity adds a source parameter value to a target list
- the set activity sets the target parameter to the new value of the source parameter
- the remove activity which only has a target parameter should remove the according feature from the target model
- the replace activity should replace the target parameter feature with the source parameter feature

However, since then, the ambitions for the future of the Weaver have changed and it was decided that the activities should be an element that fetched the input values and stored output values. Therefore, the activities (or what was meant for them to become, more like) became empty containers in which to put some functional code (whether it be Java or some other language was still to be determined). Therefore, the whole Weaver became a framework that allows to connect “customizable” events and activities. As a matter of fact, the AbstractActivityAdapter has but one abstract method, the doExecute() method (called by the execute() that first verifies that the activity can indeed be executed) which is to be implemented by the user’s customized activities.

\(^1\)That subject is discussed in the next chapter

46
In that light, a valid idea would be to want to use an already existing and approved transformation language like ATL inside those empty activities. It seems to be the most promising of all transformation language: it is the most widespread and it is already integrated in Eclipse. The eclipse platform provides a support for ATL in the form of a series of plugins available on the update site under “modeling”. That plugin provides a fairly simple way to transform models using ATL. It’s as simple as creating a new ATL project, providing it with two `.ecore` metamodels, creating the transformation rules in an `.atl` file and clicking “Run”.

Therefore, it should technically be possible to launch such an ATL transformation from Java (i.e. from another Eclipse project in our case). The way to go could be to let the user define the role of his activities in ATL. Each activity would then be executed by calling the ATL plugin.

However, the activities’ source and target parameters would respectively have to reference the very roots of the input and output.ecore metamodels. Defining location paths to particular features of the model would indeed make no sense since ATL rules work on “absolute” metamodels and define the concerned features themselves. Also, technically, the graphically represented “source” and “target” parameters of the activity would in fact not be those of the activity itself but rather those of the ATL rule that the corresponding activity is to execute.

### 4.3 Transformation semantics

Semantics is defined by the Cambridge dictionary [8] as “the study of meaning in languages”:

*“Syntax describes the rules by which words can be combined into sentences, while semantics describes what they mean.”*

The objective in this case is to propose a representation for transformation processes on top the binding model provided by Wazaabi. The point of this representation is that it conveys the meaning, the semantics of the process. It should portray what happens in the transformation process so that the purpose of said process can easily be understood by a user.

Olivier Moises, developer of Wazaabi, led the way to choosing a representation by opting to use BPMN for the modeling of transformation processes. Our study of other technologies that involve transformation along with the approach used in Wazaabi tends to prove that the choice of using BPMN was a well-advised one. The choice of dividing transformations into smaller, independent units seems like a valid one when looking at other technologies involving transformations of any kind (cf. Chapter 2). We have seen that, in other aspects of data transformation, it is common to represent transformations as a flow of successive predefined operations. In both environments that were introduced in Chapter 2 (namely Talend Open Studio and TIBCO Business Studio), transformations are composed of atomic operations that flow into each other from start to finish according to paths laid down by the arrows that link the units together.

We have also observed in that chapter that the concept of describing sets of matched rules to define transformations is becoming a “de facto” standard whether it is for transforming XML documents (XSLT) or actual models in the MDA sense of the word (ATL).
4.3. TRANSFORMATION SEMANTICS

The division of transformation into independent units accurately depicts that concept. In XSLT and ATL, the rules are independent modules. They execute a certain operation on a specific part of the source data, regardless of other rules. These rules can occur in their own time or be called by other rules. The analogy between this concept of rules and the graphical representation of flows is illustrated at Fig.4.1. In this example, the first rule is matched when a model object \( \text{EObject} \) is encountered in \( \text{ecore} \) metamodel. The execution of the first rule triggers that of the second rule.

However, in this example, the arrow does not have the same meaning as in Talend Open Studio or TIBCO Business Studio. In these two environments, the arrows have a chronological meaning. They describe in what order the operations are to be executed. This is not what the arrow means in the example at Fig.4.1. The latter rather represents the passing of a parameter from the first rule to the second which is why the arrow is that of a message flow in BPMN. If the \text{getCross} rule was a regular ATL matched rule (not a lazy one, that is), then Fig.4.1 would change into Fig.4.2. This reflects the fact that the concept of time is absent in ATL. All matched rules will be run regardless of any order or chronology once the transformation is initiated. If we were to simplistically model a generic ATL process using BPMN, we would have a similar result as that shown in Fig.4.3. This diagram illustrates that the only element that is dependent on time is the “start” event (which occurs in Eclipse when the user clicks the “Run” button). Upon occurrence of the “start” event, the whole ATL file will be processed and all the rules, if matched, will be executed. This goes also for the representation semantics in ETL, where the only event (the “start” event) is when the “Run” button is clicked, and for TIBCO, where the only

\begin{verbatim}
rule Example {
    from
    s : ecore!EObject
    to
    t : metamodel!Node {
        name <- s.toString(),
        edges <- thisModule.getCross(s)
    }
}

lazy rule getCross {
    from
    i : ecore!EObject
    to
    rel: metamodel!Relationship {

    }
}
\end{verbatim}

(a) An example of a rule (Example) calling another rule (getCross) in ATL

Figure 4.1: Illustration of the analogy between the concept or matched rules and the common graphical representation for transformation

(b) Representation of Fig.4.1a using notations common to the technologies introduced in Chapter 2
CHAPTER 4. ANALYSIS OF THE BINDING MODEL  

4.3. TRANSFORMATION SEMANTICS

(a) A modification of the ATL code of from Fig.4.1a where the second rule is not a lazy one anymore

(b) Graphical representation of Fig.4.2a

Figure 4.2: Modification of Fig.4.1 to remove the “laziness” of the second rule

Figure 4.3: Simple BPMN representation of an ATL process
event (also the “start” event) is the reception of a message by the mediation interface. In all those technologies, no external events can affect the flow of the process once it has been started.

This absence of temporal dependency is what ATL lacks in order to allow incremental transformation and what the Wazaabi Weaver provides in order to attend to that issue. The mechanism of listening to events present in the Weaver adds indeed a temporal dimension because it specifies that “if at any given moment in time this particular event should occur; then this particular transformation is to be executed”. Of course, the word “time” here means the execution time, the lifecycle of the binding model instance.

We have established the merits of representing transformations as a “BPMN-like” flow of activity. If we now add the extra factor in the Weaving Model that is the possibility of an external event affecting the process’ execution, we end up with a flow of events and activities. This goes to prove that the choice of BPMN as transformation semantics for the Wazaabi Weaving model is only natural and accurately depicts what happens during the transformation.

4.3.1 Extending BPMN

We now know we have a good, adapted representation semantics for transformations in the Wazaabi Weaver in the form of BPMN. Its purpose is to model business processes but we have shown it has everything the Weaver needs. The question we should now ask ourselves is this: “couldn’t we extend or adapt the concept used in BPMN to make a more “model transformation”-oriented process modeling notation? We answer that question with the affirmative. First of all, the sequence flow are obviously conserved “as is” since there semantics remains strictly the same in the Weaver as in BPMN. Now, indeed, we can safely assume that any given activity intended to perform model transformation will take some “inputs” and “outputs” attributes and, therefore, we could validate the extension of the BPMN activities as suggested at Fig.3.6b. The addition of attributes to the activities contributes a lot to the semantics of the notation. It also gives much more sense to a binding between a parameter and an EMF resource.

However, because of the will to make the activities empty containers for transformation code and because, to this day, no restriction have been expressed as two the type of code that should be allowed to be executed by activities, one could argue that we cannot assume with absolute certainty that such representation will be compatible with every (transformation) language. For instance, we could allow an activity to have multiple input parameters because it could very well represent the reality of a transformation module in Java. However such representation would be inconsistent if the activity was to be “filled” with an ATL rule which can only accept one source parameter.

There is less uncertainty surrounding the events. These are not involved in the transformation per se. Their main role is to receive EMF notification from model objects. The model objects as well as the notifications are Java classes. There is therefore no reason for wanting to use some other language than Java atop them. We can also assume that the only event involved is the modification of a structural feature of a model (“Changed” event) with the addition of the “Refresh” event (see third “line” of Fig.3.12b). The “Refresh” event occurs (usually at the user’s request) when the whole target model needs to be updated. This event occurs at least once, at the very instantiation of the source
changed events can be represented as in fig.4.4a. the bpmn event symbol is extended to signify a change. the asterisk is universally used to represent that the state of something has changed since the last time it was validated. no further than in eclipse for instance, the asterisk \( \ast \) is prepended to the name of a file that was modified since it was last saved (fig.4.4b). this goes also for most text editors. additionally, \( \text{changed} \) events in the weaver should have an attribute that is the location path of the observed structural feature.

refresh events can be represented as in fig.4.5a. the bpmn event symbol is extended to signify that \( \text{refresh} \) occurred. the \( \text{two head-to-toe arrows} \) symbol is universally used to represent the \( \text{refresh} \) action. again, no further than in eclipse, the same symbol is used in the workbench to represent the very same action (fig.4.5b). that same symbol is also used on the \( \text{refresh} \) button of major web browsers. these events do not have any attributes.

lastly, an element that has been overlooked up until now and that should (and can) have a semantic representation are the \text{emflocators}. these are a constant in the wazaabi weaver and their meaning is clear. as explained in section 3.3.4, they are used to physically locate an emf resource. bpmn has an element that is semantically very close to the \text{emflocator} and that is the “data object” (fig.2.15a). in the wazaabi weaver, they will have a single \( \text{uri} \) attribute. of course, this implies that the representation of the “binding” element (a simple black line, unnavigable) has to be formally added to the representation semantics.
4.4 Conclusion

In this section, we analysed the work that is the Wazaabi Weaver. We showed that it solves the problem of the incremental transformation, that all other model transformation technologies lack, by establishing a mechanism of listening to events at runtime. We also have shown that the Weaver is but a framework for doing incremental M2M transformation and that other technologies or transformation programs can be “plugged” on it. Finally we have established a representation semantics that more accurately portrays transformation with a reserve on the activities due to their nature of “blank slates”.
Chapter 5

Specification of the Binding Editor

The purpose of defining an adequate representation semantics for transformation in the Weaver is to provide an efficient graphical editor for declaring bindings. This chapter will focus on developing such an editor.

The objective here is to create a graphical editor that provides for declaring model bindings. This editor has to implement the semantics proposed in the previous section. In other words, this editor will allow us to “draw” binding processes by connecting events and activities together through simple mouse clicks.

Upon declaration of the transformation, the editor should generate the corresponding model objects and implement the specified binding processes. Essentially, the aim is to provide a more elegant and more intuitive substitute for the Ecore editor provided by EMF (as shown in Fig3.12b for instance).

In this chapter, we will provide a complete and formal description of the expected product [29]

5.1 Purpose of the product

5.1.1 Background

The Wazaabi Weaver is an effort to provide a framework that enables to perform bindings between live models. The step further consists of provided an intuitive user interface to declare those bindings. However, the developers lacked the semantics to efficiently represent such bindings. This work addressed this issue and what remains is the implementation of an interface.

5.1.2 Goals of the project

This editor needs to provide an easy way to graphically create instances of the Wazaabi Binding Model in order to specify transformation rules between EMF models.
5.2 Mandated constraints

5.2.1 Solution constraints

The binding editor needs to be a means to instantiate the Wazaabi Binding Model and therefore be able to work with this model.
No modification can be done on the core Binding Model.
The editor needs to present the data according to the representation semantics discussed in previous chapters.
The editor needs to persist the created models in the same format as expected by the Wazaabi Binding Engine.

5.2.2 “Off the shelf” packages

Because the model at the base of the editor is an EMF model, the editor will require the EMF plugin in order to run.

5.3 Naming conventions

Binding element – any atomic subelement of a binding process
Binding process – a sequence of events and activities linked together by sequence flows that specify the transformation process from one (or more) structural parts of one model to one (or more) structural parts of another model (see Chapter 3)
Node – any element that has incoming and/or outgoing edges (i.e. sequence flows). Can be instances of either events or activities (see Chapter 3)
Event – something that happens at some point during runtime and is a trigger for a binding process or a part of it (see Chapter 3)
Activity – an action that needs to be taken in the course of a binding process
Wazaabi Weaver – the name of the complete Wazaabi project that includes the Wazaabi Binding Model, the Binding engine and the hereby discussed editor
Binding Editor – the desired product

5.4 Scope of the work

5.4.1 Current situation

As of now, the way to create instances of binding process is to use the editor generated by EMF. As we have seen previously, the EMF-generated editor presents the model as an SWT tree which carries no semantics. The aim is not to modify and/or enhance this EMF-generated editor and the new editor shall therefore not supplant it. The generated editor shall still be usable.
5.4.2 Context of the work

The work context diagram presented at Fig.5.1 aims to identify all relevant inputs that comes in the work and all the data that is ouput by it [31]. The boundary of the work is the outline of the central circle. It shows external entities that we need to understand and be able to work with in order to efficiently design the editor.

![Work context diagram for the binding editor](image)

Figure 5.1: The work context diagram for the binding editor[31]

5.4.3 Work partitionning

The following is a list of business events to which the work reponds. For each event, this table lists its name, its inputs and output and a summary of the business use case.

<table>
<thead>
<tr>
<th>Event name</th>
<th>Inputs &amp; Outputs</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. User loads model in editor</td>
<td>Binding model (in)</td>
<td>Create model objects.</td>
</tr>
<tr>
<td>3. User modifies element properties</td>
<td>New values (in)</td>
<td>Ask binding model for new instance of the object.</td>
</tr>
<tr>
<td>4. User modifies constraints of a figure</td>
<td>New dimension and/or position (in)</td>
<td>Display new object on editor window.</td>
</tr>
<tr>
<td>5. User selects element in editor window</td>
<td>Mouse event (in)</td>
<td>Modify values in the model.</td>
</tr>
<tr>
<td>6. User saves the model</td>
<td>Binding model (out)</td>
<td>Update property values displayed in editor.</td>
</tr>
</tbody>
</table>

Table 5.1: The business event list [31]
5.5 Scope of the product

5.5.1 Product boundaries

The use case diagram at Fig.5.2 illustrates the boundaries of the editor. The only actor is the user of the editor and the use cases reflect the possible interactions between a user and the editor.

5.5.2 Use case list

Here are the description of the individual use cases represented at Fig.5.2. The structure used for these use cases conforms to that recommend in the Volere template [30].
CHAPTER 5. SPECIFICATION

5.5. SCOPE OF THE PRODUCT

Use Case 1

Actor

End-user

Description

Create a new binding process

Fit criterion

The user shall be able to create a new file that is editable by the Binding Editor

Use case scenarios

The user creates a new instance of the binding model.
The new model is opened in the Wazaabi Binding Editor as a blank page.

Use Case 2

Actor

End-user

Description

Adding a node to a binding process

Fit criterion

The editor will provide a way for the user to graphically create a new node in the binding process, whether it be an Activity (2.1) or an Even (2.2).

Use case scenarios

The user initiates an action to create a new node in the process.
The editor instantiate a new object of the requested type in the underlying model.
The editor adds the image of the new object to the current editor window.
Use Case 3

Actor

End-user

Description

Linking together two nodes of the binding process

Fit criterion

The user will be able to graphically link a node to another. This will result in the model accurately reflecting the user’s request.

Use case scenarios

The user initiates an action to create a link that goes from a source node to a target node.
The editor instantiate a new SequenceFlow object in the model.
The attributes of the targeted elements (one edge and two nodes) are updated accordingly.
The editor displays an arrow that links the source node to the target.

Use Case 4

Actor

End-user

Description

Editing the properties (features) of a particular binding element.

Fit criterion

The editor will provide a way for the user to edit the values of an element’s features that are not graphically represented.

Use case scenarios

The user selects the element he wishes to modify a feature from.
The user inputs the new value for that feature in the editor.
The editor updates the value for the corresponding feature of the corresponding model object.
The editor displays the feature’s new value in place of the old one.
Use Case 5

Actor

End-user

Description

Importing an EMF model that will be the source or target of a binding process

Fit criterion

The user will have a way to select any EMF model that he wants to bind to his process.

Use case scenarios

The user selects, in the editor, the EMF model he wants his process to be bound to.
The EMF model is added to the editor who creates a new Locator object that points to the selected resource.

Use Case 6

Actor

End-user

Description

Binding a node to a feature of an external EMF model.

Fit criterion

The editor shall provide a way that the user is able to specify a binding between a EMF resource (model) and a node’s parameter (see Chapter 3)

Use case scenarios

The user initiates an action to create a link that goes from a source node to a target resource.
The editor instantiate a new Binding object in the model.
The parameters and elementLocator attributes of the Binding model objects or set accordingly.
The editor displays a line that links the node to the resource.
CHAPTER 5. SPECIFICATION

5.6. USABILITY REQUIREMENTS

Use Case 7
Actor

End-user

Description

Persisting the binding process model on disk.

Fit criterion

The user will be able to store all the information on his model in a way that it can be loaded another time without any loss of information.

Use case scenarios

The user chooses to save his work in the editor.
The editor overwrite the content of the file with the current state of the model.

Use Case 8
Actor

End-user

Description

Loading an existing Binding Process into the editor.

Fit criterion

The user will be able to open an existing Binding Process model in the editor which will display the content of the file using the appropriate representation semantics.

Use case scenarios

The user selects a Binding Process model and chooses to open it in the Binding Editor.
The editor loads the all the information, creates all the model objects and displays the Binding Process model in the editor window.

5.6 Usability requirements

5.6.1 Ease of use

The editor shall be usable by a user with no programming experience but an understanding of business processes.
It shall be usable by people with little understanding of English.
The editor’s interface shall be intuitive.
The editor should use editing concepts as common as possible: if a similar user action is usually performed in a same way in most editing environment then the binding editor should try to comply to that norm.

5.6.2 Ease of learning

The editor will require next to no time to learn provided the user has previously studied the Wazaabi Binding Model.

5.7 Non-functional requirements

5.7.1 User interactions

The editor should provide a palette of atomic elements that the user can drop on the editor window to add them to the process.

The user should be able to link an element to another simply by clicking on the two elements consecutively after having selected the appropriate linking tool.

The user should be able to reshape the graphical elements as he would in most other editing environments (by clicking on a side/corner of the shape, dragging it to the desired position and then releasing it)

The user should be able to delete model elements independently, by pressing the delete key for instance.

The user should be able to import a source or target model in the same way that he adds elements to the process. The user adds a new element that represents a source/target model into the editor window. This element needs to be in relation with an existing EMF metamodel. The user could, for instance, be prompted with a wizard window that allows him to select an existing metamodel on his file system.

The user should be able to bind a node to a model feature in a similar way as for linking nodes together. The user should be able to do so by selecting a “binding” tool and clicking on the node and the model resource consecutively. The user should then be able to specify to which type of the node’s attribute this binding applies (typically source or target for most activities) as well as the path to the desired model feature in the bound model.

5.8 Conclusion

We have hereby formally specified the requirements that a Wazaabi Binding Editor should fulfill based on our research in the previous chapters of this work. We have listed the use cases that the editor can be subject to and the user actions that it should be able to respond to.
Chapter 6

Implementation of the Editor

Now that we have laid down the specification requirements for the binding editor. We can start implementing it. The first decision to make is the choice of the technology that we will use to build our editor. Because every Wazaabi component exist in the form of an Eclipse plugin, it is only natural that our Binding Editor also be an Eclipse plugin. Eclipse essentially offers two tools for building graphical editors: the Graphical Editing Framework (GEF) and the Graphical Modeling Framework (GMF). The latter allows to automatically generate EMF- and GEF-based graphical editors only by defining models. The framework needs four EMF models: one domain model (which in this case would be the Wazaabi Binding Model), one model that defines the graphical representations, one model that specifies the editing tools provided with the editors and one model that maps elements of the three previous models together. Once provided with those four models, the framework can automatically generate the editor that allows to manipulate the domain model.

For the purpose of this work we will choose to implement our editor with GEF rather than GMF. Although creating editors using GEF is slightly more lengthy and complicated than with GMF, it is of more interest to us in the scope of this work for it will allow to investigate the design and architectural principles behind this framework (and therefore, our editor). An editor generated using GMF is, in fact, built on GEF framework but its implementation and architecture are hidden from the developer. Furthermore, using GEF will allow us to really customize the “view” part of the editor which has been the focus of this work.

6.1 Introduction to GEF

The Graphical Editing Framework (GEF) is a very complete and complex toolset for building graphical representations of existing models. It allows to develop feature rich graphical editors. The editing possibilities of such editors extend from changing properties of model objects to graphically building model instances from the ground up. GEF-built editors support all common actions such as Drag’n Drop, copy/paste, undo/redo, and more. The GEF plugin contains the core mechanisms along with a variety of classes (mostly abstract) that can and should be subclassed by the developer to conform them to its particular needs. In other words GEF defines the bone structure and core functions of the
editor and it is up to the developer to complete it with the components and behaviours that are specific to his needs.

Because GEF relies on another plugin (Draw2D) for all its representation purposes, we will start by introducing the latter before diving into GEF.

### 6.1.1 Draw2D

Draw2D is a lightweight graphical system that GEF depends on for its display. Draw2D is a toolkit of graphical components called figures [35]. Lightweight means that it operates at a level removed from the operating system and is hosted by a single heavyweight control. The figures are pure Java and have no corresponding resource in the system. The base class in Draw2D is the `LightweightSystem`. The `LightweightSystem` is associated to an SWT canvas and will hold the composition of figures. The `LightweightSystem` itself has no visual representation but handles the interactions with the exterior. It will be responsible for dispatching events such as mouse clicks to the EventDispatcher or paint events to the UpdateManager (see Fig.6.1).
Figures

The figures are the main component in Draw2D and are treated as if they were normal windows: they can have focus and selection, be the target of mouse events or be moved and resized. Figures are added to the LightweightSystem via parent-child relationships between each other. All of the Draw2D GUI components extend the main Figure class. Fig.6.2 shows a small subset of Draw2D figures. We can distinguish three main categories among those figures.

Shapes (subclasses of the Shape class) implements geometrical figures instantiated based on a couple parameters. Such shape classes include Rectangle, RoundedRectangle, Ellipse, Triangle, etc.

Widgets are figures that provide the user with some control on the input. Draw2D widgets include the checkbox, the button, the text entry field and more.

Layers and panes are figures that will serve as containers for other figures. They provide scaling, scrolling and laying out of their children figures.

Methods

The Figure class has many methods for handling its properties. They can:

Manage their visual aspects by setting their size and location. Maximum and minimum sizes can be defined as well as foreground and background.

Register event listeners for all standard SWT events (addMouseListener(), addKeyListener(), addFocusListener(), etc) and for custom events (addPropertyChangeListener).

Manage their hierarchy by adding children figure. In Draw2D, any figure can be a parent to multiple children figure or be the child of a parent figure.
Managing graphics. Draw2D provides a Graphics class - similar to SWT’s GC class - which figures will use to display their representation. By calling the paint() method, they can specify how and what the Graphics object should display within their bounds.

Layouts and layers

LayoutManager classes are used by figures to manage the positioning of their children figures. For instance the XYLayoutManager requires figures to have a constraint of type Rectangle that specifies its size and coordinates whereas the FlowLayoutManager into rows or columns.

Connections

Connections are a special type of figure that represents a link between two Draw2D figures. A connection figure has the particularity to not be owned by a parent figure but instead by its source and target. For a figure to be able to be the source or target of a connection, it has two own one or more connection anchor. These anchors specify the point on the source or target figure where the extremity of the connection is to be attached.

6.1.2 The GEF framework: MVC architecture

While Draw2D focuses on efficient painting and layout of figures, GEF adds editing capabilities to it. The framework works according to the Model-View-Controller (MVC) design pattern. This implies that the application is conceptually separated in three parts (Fig.6.3 shows the MVC separation for the GEF plugin):

The Model is the data that gets persisted and holds all the information on the business elements. At this point in this work, we know what a model is. In standard use of GEF, models or hand-coded Java classes but it does not necessarily have to be so. A GEF application can very well work with an EMF-generated model as will be the case for our sample editor.

The View is the part of the application that is visible to the end-user.

The Controller makes the link between the model and the view and acts as the application’s interface for the user. The controller part will be responsible for all the editing.

Since the Model implementation is entirely up to the developer and the View is handled by Draw2D, the GEF library defines the controller part of the application. The main element in GEF is called the EditPart. In GEF each editable component is composed of three elements: a Draw2D figure (the View), its corresponding Java object from the model and an EditPart that implements the controller for this component.

6.1.3 EditParts

EditParts are the links between model objects and their graphical representation. Because in a good MVC implementation the Model and the View do not “know” about each
Figure 6.3: The MVC architecture of the Graphical Editing Framework [35]

Figure 6.4: EditParts and figures usually share a structure similar to that of the model [35]

other, only the EditPart knows of its corresponding model object and Draw2D figures. An EditPart is associated to a model object through its `setModel(Object model)` method an to a figure through its `createFigure()` method.

EditParts usually form a structure similar to that of the corresponding model. If an EditPart controls a model object that has a containment relation with some children model object, then this EditPart will be the parent of the EditParts that control those children model object. And in the same way. The figure that represents the “container” model object will be a parent to the figures representing the children model object (see Fig.6.4. The only exception to this generalization comes with connections. Indeed, no matter how connections are stored in the model, ConnectionEditParts are managed by their source and target EditParts, in a similar way as for connection figures. A well designed GEF application should have a top level EditPart, called Contents EditPart. This EditPart will be a parent to all other EditParts of the model. It will usually have no visual representation and its Figure will only consist of a Layer for the children Figures to be laid out in.
There are three types of EditParts: GraphicalEditParts, ConnectionEditParts and TreeEditParts. The third type, which is used for building trees of the model, is virtually never used in the editor, only with the outline view if provided. As for the other two, the ConnectionEditPart will manage connections whereas the GraphicalEditPart will manage every other model element. The responsibilities of an EditPart are to create and maintain the view, children EditParts and connection EditParts as well as support editing of the model.

### 6.1.4 Requests

Requests or the objects that are used by GEF typically when the user interacts with the GUI. They contain information necessary to perform the requested action. There are several types of requests, used in different scenarios. The three main types are:

**CreateRequest** is used whenever a new model object needs to be created. This request contains, among other, information about the new model object to be created (e.g. the new object’s type or a String that maps to it).

**GroupRequest** is a type of request that can apply to multiple EditParts at once. The affected EditParts are retrieved with the method `GroupRequest.getEditParts()`.

**LocationRequest** is a request that keeps track of the exact location, for example, of the user’s click.

All other requests extend one of those three. Here are some examples:

**ChangeBoundRequest**, as its name implies, is instantiated when the location and/or size of EditParts need to be changed. This request is a specialization of the GroupRequest as the bounds of multiple EditParts can be changed at once.

**CreateConnectionRequest** is a subtype of CreateRequest that is particular to connection. In addition to knowing the type of the object to be created, it needs to provide the source and target EditParts for this connection.

**SelectionRequest** simply is used when selecting an EditPart. It naturally extends the LocationRequest, not only because the selected EditPart is the one situated at the location of the user’s click, but also because the EditPart may have a different behaviour depending on the region it was clicked on. The request possesses various methods to signal any button (mouse or keyboard) being pressed while the EditPart is selected.

### 6.1.5 EditPolicies

EditParts do not handle editing themselves. Instead, they delegate to the EditPolicies. EditPolicies are installed on EditParts upon their creation by calling `createEditPolicies()`. EditPolicies are categorized by role. For each role, an EditPart may have up to one EditPolicy. For each role, GEF provides a basic EditPolicy. These EditPolicies are functional.

---

1 There can very well be multiple types of connection. For instance, in an UML diagram editor there will be different connection Figures and EditParts for associations, composition, aggregation, etc.
but are often extended to adapt to the needs of the application. Some of the most important roles are:

**Component role** is the main role in GEF as it applies to all components, graphical or not, even the contents EditPart. The ComponentEditPolicy handles operations that involves the model element of an EditPart (e.g. deletion).

**Container role** is responsible for handling operations typical of a container such as the creation of children.

**Connection role** is the equivalent of the Component role for connection EditParts.

**Layout role** is useful if the container EditParts has an associated Layout. If so this role handles operations specific to said Layout.

**GraphicalNode role** is used for elements that act as nodes, i.e., that can be the source or target of connections. The associated EditPolicy will therefore handle the creation and deletion of connections.

### 6.1.6 Commands

Commands are the elements that actually modifies the model. They are created by the EditPolicies when they are requested to do so. Commands need to be implemented by the developer and extend the abstract Command class. Its main methods are `execute()`, `undo()` and `redo()`, which need no explanation and must be implemented and instantiated by the developer. The typical commands one will find in a GEF editor are the command to create an element, the command to delete one and the command to change its position and/or size.

When an action is taken by the user, a Request object will be created and sent to the EditParts which will, in turn, forward them to their EditPolicies. If, for instance, the Request is a CreateRequest, then the EditPolicy will be asked `getCreateCommand(Request request)` which returns a `Command` object. This method, implemented by the developer, will then instantiate a new create the new creation command and give it the type of the new object and its location based on the information provided by the request.

The overall course of events is this [27]:

1. The user interacts with the GUI using his mouse or keyboard.
2. This results in events that are forwarded to the currently selected tool.
3. These events are interpreted by the tool to generate the appropriate Request.
4. The Request is sent to the EditParts who forward them to their EditPolicies.
5. The EditPolicy creates the command that satisfies the Request and sends it back.
6. Once the tool received a command from an EditPart, it can execute it through the command stack. The command stack is an undo-redo stack.

68
In a good GEF design, commands should only know about the model and, therefore, only manipulate model objects. A creation command will essentially include the new model object to the existing model, add the new model object to its parent’s list of children and specify its position and size. A command to change the constraints of an element will update the values of the corresponding model object. In the latter case, the model object is obtained by calling `getModel()` on the EditPart that was the target of the action.

### 6.1.7 Reflecting model changes

Because, when an action is taken by the user, only the model is modified, there is a need to implement a mechanism that updates the view accordingly. Therefore, when the feature of a model element changes\(^2\), the EditPart that controls said model elements has to be aware of it. In most cases, the EditPart will call its `refreshVisuals()` method to update the figures aspect. If the feature relates to connection and the EditPart implements GEF’s `NodeEditPart` interface, then it is the connections that will be refreshed instead.

### 6.1.8 Palette

The PaletteViewer displays the editing tools that are made available to the user. In addition to the selection tool, and the group selection tool (or marquee), the palette will (usually) contain one creation tool per type of editable graphical component. When the user clicks on a tool, it is then said to be the active tool. Any event that occurs in the viewer is forwarded to the active tool that creates the appropriate request. For example, if the active tool is the creation tool for a given element, then the event of a mouse click will result in the creation of a `CreateRequest` for said given element at the location of the click.

### 6.2 Building the Binding Editor

GEF editors are built as Eclipse plugin projects. This means that the build of the project is packaged as a fully functional Eclipse plugin that can be injected into an Eclipse application instance. An Eclipse plugin project needs to be specified the main class to be launched upon activation of the plugin. In any case, this main class must be the one initializing the graphical editor. In our case this class is called `BindingEditor`. There are a few possibilities as to how the editor will be displayed. Mainly, it can be displayed in a standalone window or it can be included as a view in the Eclipse workbench. We chose the second possibility.

In all generality, the main editor class should in some way subclass `org.eclipse.ui.part.EditorPart`. We chose to have our `BindingEditor` extend `org.eclipse.ui.part.MultiPageEditorPart` which is a particular Eclipse editor view that can have multiple tabs (pages), each of them containing an editor. The editor class contains a lot of methods and its implementation is fastidious. A full explanation of its functionalities is not interesting for the purpose of this work.

\(^2\)The addition of a new element to the model is a change to the parent element’s list of children.
6.2.1 Managing the model

The development of a GEF application is independent of the type of model it is built upon. Indeed, the EditPart’s `setModel()` method takes a Java `Object` as an argument. The model used in our application is the Wazaabi Binding Model which was generated with EMF. Doing so has a few consequences seeing that, in our case especially, we cannot modify the EMF generated code.

Listening for model changes

We do not have the option to implement the typical `firePropertyChange()` type of methods in our model objects for our EditParts to listen to. Fortunately, we have explained in chapter 3 that EMF provides a notification mechanism that every model object implements. By having our EditParts implement the `Adapter` interface of EMF, we can register them to their corresponding model objects as shown in example 6.2.1.

**Listing 6.2.1** The `NodeEditPart` is an Adapter for its `Node` model object

```java
public class NodeEditPart extends AbstractGraphicalEditPart implements Adapter{

    public void activate() {
        if (isActive())
            return;
        ((Notifier) getNode()).eAdapters().add(this);
        super.activate();
    }

    public void deactivate() {
        if (!isActive())
            return;
        ((Notifier) getNode()).eAdapters().remove(this);
        super.deactivate();
    }

}
```

Figure constraints

As we have explained earlier, it is part of GEF’s design principles that commands only know about model objects. Whenever a modification is made, it is made to the model. The controllers of the modified model objects are made aware of the change and they proceed to update the view accordingly. This design choice implies that every information needed to represent the model is stored in the model. So, when the user selects an object in the editor window and proceeds to change its size or location, it results in the selection tool sending a `ChangedBoundRequest` to the Editparts which, in turn, create the command that will modify the model. This command would have to change the model object data that relates to its position and/or size. The problem in our case is that this particular information is not part of the model. This is but natural considering that the model is that of a (business) process and the definition of a process is not in any way dependent on the way it is laid out on an editor window.

However, for our editor to be user-friendly, it should allow the user to lay his binding process out as he wishes. Therefore, our editor needs to be provided with this information.
Figure 6.5: Extension of the Binding Model for the purpose of the editor

one way or another. Because we cannot alter the Wazaabi Binding Model in any way, the first and easiest solution that comes to mind is to delegate to store this information in the Node’s controller (the NodeEditPart). However, this violates the design principles of GEF. Firstly, it would be a “dirty” way to solve the problem for it is not the place of an EditPart ot provide such information. The role of an EditPart should remain to control the interactions between the model, the view and the user. Secondly, it would imply that the commands would have to manipulate EditParts in order to provide them with this information which is again in contradiction with GEF’s principles.

The other solution and the one we opted for is to create an extension of the Binding Model which will only be used for the purpose of the editor. Doing so respects both the GEF principle that all information should be stored in the model and the requirement that the Binding Model no be modified. The models extension is represented at Fig.6.5. We have created a new object NodePositionned that extends the node from the original binding model. That way, the node used in our editor is still, by any standards, a Node in the Wazaabi sense of the word\(^3\) and, at the same time, it solves our problem by having a position and dimensions. It is interesting to note that it is not sufficient for the other elements (Events, Activities and corresponding subclasses) to just extend this new NodePositionned type. Indeed not only does an ActivityPositionned have to inherit the position and dimensions attributes from the NodePositionned, it also has to extend the original Activity\(^4\) in order to be recognized as such. Fortunately, it is not a problem thanks to EMF’s support for multi-inheritance.

\(^3\)org.wazaabi.model.bindingcore.process.Node
\(^4\)org.wazaabi.model.bindingcore.process.Activity
6.2.2 Implemented use cases

Creating a new Binding Process

Every GEF application (and all Eclipse plugins for that matter) possess a configuration file called `plugin.xml`. This file contains, among other information, the file extension that will be opened in that editor. To create a new resource that shall be editable by the GEF editor, the user simply creates a new blank file with the appropriate extension. He is then able to open this file with the desired editor.

Listing 6.2.2 BindingEditorMethods for initializing the Binding Editor

```java
public class BindingEditor extends MultiPageEditorPart {

    public void init(IEditorSite site, IEditorInput input) throws PartInitException {
        try {
            IFile file = ((IFileEditorInput) input).getFile();
            process = create(file);
            if (null == getProcess())
                throw new PartInitException("The specified input is not a valid network.");
        } catch (CoreException e) {
            throw new PartInitException(e.getStatus());
        } catch (ClassCastException e) {
            throw new PartInitException("The specified input is not a valid network.", e);
        }
        super.init(site, input);
        getDelegatingCommandStack().addCommandStackListener(getDelegatingCommandStackListener());
        getSite().getWorkbenchWindow().getSelectionService().addSelectionListener(getSelectionListener());
        createActions();
    }

    private BindingProcess create(IFile file) throws CoreException {
        BindingProcess process = null;
        modelManager = new BindingProcessModelManager();
        if (file.exists()) {
            try {
                modelManager.load(file.getFullPath());
            } catch (Exception e) {
                modelManager.createProcess(file.getFullPath());
            }
        }
        process = modelManager.getModel();
        if (null == process) {
            throw new CoreException(new Status(IStatus.ERROR, BindingEditorPlugin.PLUGIN_ID, 0,
                "ERROR loading the process.", null));
        }
        return process;
    }
}
```

Code sample from Listing 6.2.2 show the main method from the `BindingEditor` class that are responsible for initializing the instance of the editor. In a nutshell, we can see that the `init()` method (which is invoked by the GEF framework) attempts to retrieve a file
from the IEditorInput argument and tries to create the top level BindingProcess model object from that file. In turn, the create() method attempts to load the resource from the file or, if that fails, create a new BindingProcess model object. Here the modelManager references a class (BindingProcessModelManager) which acts as an intermediary between our application and the EMF model.

This also addresses “Use Case 8: Loading an existing Binding Process into the editor” as instead of opening a blank file in the editor, the user opens a file that has content. The internal process is the same as before. The editor will attempt to load the resource contained on the file and will instantiate the objects of the persisted binding process.

Adding a node to a binding process

As we explained earlier in our introduction to GEF, the framework provides all the internal mechanics for graphically instantiating model objects. By providing a creation tool for a specific element, the user click in the editor window after having selected the tool in order to create a new instance of the element in question. As we have already seen, this results in the creation of a request which is dispatched to the EditParts and, in turn, the EditPolicies and results in the creation of a command which, when executed, creates the appropriate model object. This is therefore just a matter of implementing the EditPolicy and creation command, the rest being the responsibility of the framework.

We first need to subclass the Command class in order to create our own command that will add a new model object to the model (Listing 6.2.3). When executed, this command sets the position and size of the new model object (child) and then adds the object to the BindingProcess’ (parent) list of children element. The new object is then part of the current model.

Listing 6.2.3 The CreateCommand’s execute method is called by the command stack

```java
public class CreateCommand extends Command {
    private BindingProcess parent;
    private NodePositionned child;
    private Rectangle rect;
    private Point pos;

    public void execute() {
        if (rect != null) {
            child.setX(rect.getLocation().x);
            child.setY(rect.getLocation().y);
            if (!rect.isEmpty()){
                child.setWidth(rect.width);
                child.setHeight(rect.height);
            } else{
                child.setWidth(100);
                child.setHeight(50);
            }
            parent.getElements().add((BindingElement) child);
        }
    }
}
```

We then need to provide our BindingProcessEditPart with an EditPolicy that will instantiate this command. Our EditPolicy should extend GEF’s LayoutEditPolicy which
is the one responsible for the creation of model objects (Listing 6.2.4). In this case the method that is needed is `getCreateCommand()` which our `LayoutPolicy` inherits from `LayoutEditPolicy`. This method, which takes a `CreateRequest` as an argument, creates a new instance of our `CreateCommand`, sets the parent as being the current top-level `BindingProcess`, sets the child as being the new object “carried” by the request and sets the constraints in a similar fashion. The `getHost()` method, which is common to all `EditPolicies`, returns the `EditPart` that “hosts” the `EditPolicy`. Since the `LayoutPolicy` is hosted by the `BindingProcessEditPart` (who’s figure is indeed a `Layout`), calling `getHost().getModel()` indeed returns the top-level `BindingProcess` model object.

Listing 6.2.4 The LayoutPolicy is responsible for creating the creation commands

```java
public class LayoutPolicy extends XYLayoutEditPolicy {
    protected Command getCreateCommand (CreateRequest request) {
        Object newObjectType = request.getNewObjectType();
        CreateCommand create = new CreateCommand();
        create.setParent((BindingProcess) getHost().getModel());
        if (newObjectType == Node.class) {
            create.setChild((NodePositionned) request.getNewObject());
            Rectangle constraint = (Rectangle) getConstraintFor(request);
            create.setConstraint(constraint);
        }
        return create;
    }
}
```

Of course, in order for the user to initiate such a Request, we must provide him with a palette that contains the appropriate creation tools. Part of the code for this palette is displayed at Listing 6.2.5. This partial code shows how a Selection tool, a Marquee tool and a creation tool for Activities is added to the Palette. It is a fairly simple process and this code is quite self-explanatory.

Now the user can graphically add new elements to his binding process and, with the notification mechanism implemented as shown in Section 6.2.1, the `EditParts` “hear” about the new model object and update the view accordingly.

### Linking together two nodes

The implementation of this use case is very similar to that of the previous one only it will require a different command and a different `EditPolicy`. In this case, the command is called `ConnectionCommand` and is partially displayed at Listing 6.2.6. Here the host `BindingProcess` is retrieved by calling for the object that contains either the source or target node. Once this `BindingProcess` is obtained, the new edge is added to its list of `BindingElements`. The command then proceeds to set the edge’s values for source and target nodes.
Listing 6.2.5 The palette contains the editing tools of the editor

```java
public class BindingProcessPalette {
    public static final String ACTIVITY_TEMPLATE = "ACT_TEMP" ...;

    public static PaletteRoot getPaletteRoot() {
        PaletteRoot root = new PaletteRoot();
        PaletteGroup toolGroup = new PaletteGroup("Binding Process Tools");
        List toolList = new ArrayList();

        ToolEntry tool = new SelectionToolEntry();
        toolList.add(tool);
        root.setDefaultEntry(tool);

        tool = new MarqueeToolEntry();
        toolList.add(tool);

        tool = new ConnectionCreationToolEntry("Sequence Flow",
                                               "Determines the flow of Events and Activities",
                                               new ModelCreationFactory(FLOW_TEMPLATE), ImageDescriptor
                                               .getMissingImageDescriptor(), ImageDescriptor
                                               .getMissingImageDescriptor());
        toolList.add(tool);
        toolGroup.addAll(toolList);

        PaletteDrawer activitiesGroup = new PaletteDrawer("Activities");
        List templateList = new ArrayList();

        CombinedTemplateCreationEntry entry = new CombinedTemplateCreationEntry(
                                               "Activity", "Something that does something", ACTIVITY_TEMPLATE,
                                               new ModelCreationFactory(ACTIVITY_TEMPLATE), ImageDescriptor
                                               .getMissingImageDescriptor(), ImageDescriptor
                                               .getMissingImageDescriptor());
        templateList.add(entry);

        activitiesGroup.addAll(templateList);

        List rootList = new ArrayList();
        rootList.addAll(toolGroup);
        rootList.addAll(activitiesGroup);
        rootList.addAll(eventsGroup);
        return root;
    }
}
```
Listing 6.2.6 The `ConnectionCommand` creates a new Sequence Flow

```java
public class ConnectionCommand extends Command {
    protected Node oldSource, source;
    protected Node oldTarget, target;
    protected Edge edge;
    protected BindingProcess process;

    public void execute() {
        process = (BindingProcess) target.eContainer();
        process.getElements().add(edge);
        if (source != null) {
            if (edge.getSource() != null && edge.getSource().getOutgoingEdges().contains(edge))
                edge.getSource().getOutgoingEdges().remove(edge);
            edge.setSource(source);
        }
        if (target != null) {
            if (edge.getTarget() != null && edge.getTarget().getIncomingEdges().contains(edge))
                edge.getTarget().getIncomingEdges().remove(edge);
            edge.setTarget(target);
        }
    }
}
```

The EditPolicy that takes care of creating that command is this time the responsibility of the NodeEditPart for it is nodes who possess the ConnectionEditPart and not the BindingProcessEditPart. The relevant methods of this `NodePolicy` are displayed on Listing 6.2.7.

Listing 6.2.7 The `NodePolicy` is responsible for creating the `ConnectionCommand`

```java
public class NodePolicy extends org.eclipse.gef.editpolicies.GraphicalNodeEditPolicy {
    protected Command getConnectionCompleteCommand(CreateConnectionRequest request) {
        PathCommand command = (PathCommand) request.getStartCommand();
        command.setTarget((Node) getHost().getModel());
        return command;
    }

    protected Command getConnectionCreateCommand(CreateConnectionRequest request) {
        PathCommand command = new PathCommand();
        command.setEdge((Edge) request.getNewObject());
        command.setSource((Node) getHost().getModel());
        request.setStartCommand(command);
        return command;
    }
}
```

Here there are two methods involved in creating a connection. The first one, `getConnectionCreateCommand()`, is called when the user has clicked on the first node with the

---

5 This is not to be confused with the case of the model where the Sequence Flow is indeed possessed by the BindingProcess.
connection tool. The method creates the command, gives it the new object provided by the Request and and sets the command’s source as being the model object of the EditPart that was clicked on. It then gives the incomplete command back to the Request as the “start command”. This command is completed when the user completes his action by clicking on the second node. This triggers a call to the other method, getConnectionComplete(), which retrieves the “start command” from the request and sets its target node as before, only the host of the EditPolicy is now the other node (the target).

The implementation of the connection creation tool is similar to other tools and the code is also described at Listing 6.2.5.

**Editing the properties of a binding element**

Among all the standard “views” that the Eclipse platform offers in its workbench, there is the Properties page (see Section 3.1.1). Because it is a necessity that the user be able to view the element attributes and edit them (if editable), it is of interest to implement this Properties page in our EditPart. This page is able to retrieve the properties to display from current elements either if these directly implement the IPropertySource interface or if they implement IAdaptable and have an IPropertySource adapter. Since, in GEF, every EditPart implements IAdaptable, we naturally choose the second option. The single method available is the getAdapter() method which, as we just said, should in this case return an IPropertySource object. We will therefore create a new object, which we shall call and refer to as EObjectPropertySource, and whose sole purpose will be to provide the properties of the model object it was associated with (see Listing 6.2.8).

**Listing 6.2.8** The EObjectPropertySource provides the properties of its object

```java
public class EObjectPropertySource implements IPropertySource {
    private EObject object;

    public EObjectPropertySource(EObject obj) {
        this.object = obj;
    }

    public Object getEditableValue() {
        return object;
    }

    public Object getPropertyValue(Object id) {
        EStructuralFeature feature = object.eClass().getEStructuralFeature(Integer.parseInt((String) id));
        Object result = object.eGet(feature);
        return result != null ? result : "";
    }

    public void setPropertyValue(Object id, Object value) {
        EStructuralFeature feature = object.eClass().getEStructuralFeature(Integer.parseInt((String) id));
        object.eSet(feature, value);
    }
}
```

Every NodeEditPart in our application possesses such a PropertySource as an attribute.

---

and returns it whenever its `getAdapter()` method is called as shown on Listing 6.2.9.

**Listing 6.2.9** NodeEditParts methods for Property page purposes

```java
public Object getAdapter(Class key) {
    if (IPropertySource.class == key) {
        return getPropertySource();
    }
    return super.getAdapter(key);
}

public IPropertySource getPropertySource() {
    if (propertySource == null)
        propertySource = new EObjectPropertySource(getNode());
    return propertySource;
}
```

**Persisting the binding process model on disk**

As do all Eclipse editors, our Binding has a `doSave()` method. This method is called by the Eclipse workbench when a save action has been taken in the workbench. This method is implemented in our `BindingEditor` class as shown on Listing 6.2.10. It essentially offers the user to create the file if it does not already exists and then calls its own `save()` method with said file as an argument. The save method will then attempt to call the `BindingProcessModelManager`'s `save()` method with the full path of the file as an argument. The `modelManager` simply uses an EMF resource to save the content of the model (see Listing 6.2.11). The method `getResourceSet`, in addition to creating a new ResourceSet, associates a new factory for XMI resources with the `.process` file extension. The `save()` method from the model manager calls `save()` on the Resource and, from there on, EMF takes care of the rest. The result is an XML file that contains all information about the model serialized in XMI and which is completely compatible with the EMF-generated editor.

```java
public void doSave(IProgressMonitor monitor) {
    try {
        IFile file = ((IFileEditorInput) getEditorInput()).getFile();
        if (!file.exists() || MessageDialogWithToggle.openConfirm(getSite(), getShell(), "Create File", "The file "+ file.getName() + " doesn’t exist. Click OK to create it.") {
            save(file, monitor);
            getMultiPageCommandStackListener().markSaveLocations();
        }
    } catch (CoreException e) {
        ErrorDialog.openError(getSite(), getShell(), "Error During Save", "The current network model couldn’t be saved.", e.getStatus());
    }
}
```
CHAPTER 6. IMPLEMENTATION OF THE EDITOR  6.2. BUILDING THE BINDING EDITOR

Listing 6.2.11 The Model Manager uses an EMF Resource to save the content of the model

```java
public class BindingProcessModelManager {
    private Resource resource = null;
    public Resource getResource(IPath path) {
        if (resource == null) {
            ResourceSet resSet = getResourceSet();
            resource = resSet.createResource(URI.createPlatformResourceURI(path.toString(), true));
        }
        return resource;
    }
    private ResourceSet getResourceSet() {
        ProcessPackageImpl.init();
        Map m = reg.getExtensionToFactoryMap();
        m.put("process", new XMIResourceFactoryImpl());
        return new ResourceSetImpl();
    }
    public void save(IPath path) throws IOException {
        getResource(path);
        Map options = new HashMap();
        options.put(XMIResource.OPTION_DECLARE_XML, Boolean.TRUE);
        resource.save(options);
    }
}
```

6.2.3 Structure of the Editor

Fig.6.6 shows a class diagram that models part of our Binding Editor. Because of the amount of classes needed to build just the simplest of GEF applications, it was necessary to leave some of them in order to keep this diagram readable. It shows some of the main classes of our application and the way they extend the GEF API.

6.2.4 Conclusion

Fig.6.7 shows a screen caption of our binding editor prototype. It shows a basic process composed of a Changed event that triggers a set of activities. On the right of the editing window is the palette of creation tools. Apart from the Selection and Marquee tool, it displays two collapsible drawers for Activities and Events as well as a tool for connecting the nodes. The nodes can be created by selecting the appropriate tool and clicking in the window or by simply dragging and dropping the tool on the editor window. In both cases, the node will be displayed at the point on the editor window where the mouse was clicked (or released when dragged and dropped). As we explained, this was made possible by extending the initial binding metamodel to provide the objects with information about their position and sizes. Below the editor window, the Properties page displays the attributes of the currently selected element. Only the properties that are not graphically represented are displayed in that page. Thanks to the fact that the model created with the Binding Editor is serialized in XMI by EMF, the model is compatible with both our Binding Editor and the EMF-generated tree editor. This means that when a model created with the Binding Editor is saved (by selecting Save from the menu bar or by pressing Ctrl+S), it can immediately be opened with the EMF editor for further editing. Fig.6.8 shows the same model as on Fig.6.7 opened with the default editor.

79
Figure 6.6: Partial class diagram of the Wazaabi Binding Editor
This editor is not yet functional as not all the use case listed in the previous Chapter were implemented. However, we have implemented the main functionalities and have laid down the basics of what the editor should be: a means to graphically instantiate the Wazaabi Binding Model using appropriate and intuitive representation semantics.

Figure 6.7: Typical binding process in the Wazaabi Binding editor

Figure 6.8: Binding Process from Fig.6.7 viewed with the EMF tree editor
Chapter 7

Conclusion

The aim of this thesis was, ultimately, to suggest appropriate representation semantics for the Wazaabi Weaver that was developed at Alcatel-Lucent until late 2009. The first step in order to make a well-educated choice in the matter was to study what already existed regarding data transformation. We therefore introduced several technologies, software and languages, each of them approaching transformation from a different angle. This allowed us to find analogies, recurrent patterns that tended to suggest “rules of thumb” in terms of transformation.

After this overview of what the world of IT has to offer on transformation, we ventured into the study of the tool that was of interest to us, the Wazaabi Weaver. This revealed itself to be a challenging task and we had to educate ourself on a couple of other frameworks prior to diving into the heart of the matter. Doing so, we learned about EMF, a very powerful tool and the flagship of Model-Driven Development. We also learned about Wazaabi, a very innovative tool for generating Graphical User Interfaces (GUIs) and we understood what a “live model” really is.

Once we had a sufficient grasp of the required concepts, we extensively studied the Wazaabi Weaver. We examined the binding model piece by piece, studied the function of each component, understood how to use it and how it works. This big preamble was necessary in order for us to aptly analyse the work done on the weaver so far and the direction in which it was headed. We showed how the notification mechanism provided by the Weaver attended to the issue that affects most, if not all, model transformation specifications, namely the incremental transformation. We also showed how, in its last days of development, the Weaver went from a model transformation utility, that aimed to provide the necessary atomic activities and events from which all transformations could be constructed, to a support framework that allowed to use existing programming languages in order to do incremental model-to-model transformation.

Finally, the goal of this thesis was achieved as we were able to skilfully analyse the legitimacy of the ongoing choice to use BPMN as well as to propose improvements to the already suggested semantics. In the matter, we can conclude that, if the Weaving model had indeed been provided with a finite set of predefined activities and events, we could have more easily specified an appropriate and unique representation for each of its component. However, considering the fact that there are no limits to what can be written in an activity, the fact that the Weaver is a “work in progress” and considering its finality has not been clearly defined; the representation semantics that we suggested carry as much
meaning and convey as much information as possible under such circumstances.

Despite that, we proceeded in giving a complete and formal specification for the Weaver’s Binding Editor. We identified its requirement functionalities by means of a rigorous and scientific approach. We laid down the architecture of the editor by investigating Eclipse’s powerful Graphical Editing Framework and we used that framework to implement a prototype for that editor based on the specification requirements.

In conclusion, this work represents a solid foundation for further development of the Wazaabi Weaver. We certainly hope that this thesis will be the impulse this unique and ground-breaking tool needed to finally come to fruition.
Bibliography


[27] T.V. Prabhakar, Aparna Gorthi, and Nitin Gautam. GEF.


